1995

Commercial Integrity, Roles And Object-orientation

Matunda Nyanchama

Commercial Integrity, Roles and Object Orientation

by

Matunda NYANCHAMA

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
September 1994

ISBN 0-315-99271-9

ABSTRACT

This thesis presents a study of realizing commercial security, as defined in the Clark and Wilson Model [CW87], using Object-Oriented (O-O) concepts. Roles, being an integral component of the this security model, form a substantial component of this work.

Commercial security is concerned more with integrity than secrecy, unlike in military systems where both secrecy and integrity are of equal concern. In commercial security, there is a designated set of operations on data which uniquely determine the relationships between data objects and the operations which update their state. Effectively, no operation, other than those assured to leave data objects in a correct state, are allowed. Authorization to data objects is achieved via user authorization to operations on the objects and hence determines user-object relationships. In executing the operations, the system must assure not only authorization, but also ensure separation of duty.

Role-based security is implied in the Clark and Wilson model in which specified operations are grouped to compose roles. This approach to protection is suitable for applications involving large numbers of users with overlapping user requirements and/or where there is a large number of objects. It presents a flexible (hence adaptive) means for enforcing differing ranges of security policies. It enforces the principle of *least privilege*, hence minimizing the risk of *Trojan horse* attacks.

Consequently, in part, this work focuses on role-based protection, formalizes the role concept and proposes a model for role organization and administration. This model, intended to ease access rights administration, is defined by a set of properties. Algorithms for role administration are presented. These guarantee the properties of the role organization model. Role-based protection is also studied with respect to traditional protection schemes. One aspect of this enquiry focuses on information flow analysis in role-based security systems; the other addresses the realization of mandatory access control using role-based protection. This involves the imposition of acyclic information flows and rules that ensure secrecy. It demonstrates the strength of the role-based protection approach.

A role is a named collection of responsibilities and functions which we term *privileges*. Execution of one or more privileges of a role facilitates access to information available via the role. Access to information is realized both via user authorization to the role and the role's privilege list. A role exists as a separate entity from the role-holder and/or the role administrator. In determining role organization, role relationships are used based on privilege sharing. This results in an acyclic role graph with roles being nodes and edges being role relationships. These relationships help us infer those privileges of a role that are implicitly defined. Analysis of this model indicates that it can simulate lattice-like models, hierarchical structures and privilege graphs.

Principles from the O-O paradigm are utilized to impose segmented access to object information. This approach uses methods to "window" an object's interface to facilitate segmented access to object data through different roles, and hence different users. By defining these methods to suit the intended functionality and associating them with specific roles, we in effect distribute the object interface to different roles and users. An object model is proposed as the basis of O-O executions. Further, in order to impose the well-formed transaction (WFTs) requirement, a transaction model is proposed that imposes transactional properties on method executions. By use of transaction *scripts* we can design executions to realize desirable outcomes.

Separation ot duty is another major requirement in the Clark and Wilson model. It requires object history for its enforcement. Our proposal ensures that objects track their history. Moreover, every execution on an object utilizes the object history to determine access and updates the history with any attempted access.

Finally, to demonstrate that we actually realize commercial security protection, we go through each of the properties in the Clark and Wilson model to show how it is realized.

However tight the lips

However tight the locks

However smart the traps

However smart the designers

However rigorous the specification

However accurate the routines

Remember

Between two exists no secret

A secret is among three, though

With two dead, one senile

ACKNOWLEDGEMENTS

My sincere thanks go to my supervisor, Dr. Osborn, for her studious guidance, encouragement and support throughout my entire period of this study. She not only proved a willing and selfless scholar but also offered her best in the administration of my programme. Dr. Mullin acted as my *de facto* supervisor in the absence of Dr Osborn. I feel deeply indebted. Like Sylvia, he too supported me beyond the academics and administrivia. I held fruitful discussions with Dr Hanan Lutfiyya at the preliminary stage of this research. Sheila Lindslay, in attempting an implementation of the graph algorithms of Chapter 5, uncovered a some errors in the original algorithms.

I would not have been in this programme without the initial support from my former employer the Kenya Posts & Telecommunications Corporation (KP&TC), the Kenyan Government and CIDA. The KP&TC supported me in my application for scholarship and kindly extended my study leave way beyond the originally agreed duration. The Kenyan Government, through the Directorate of Personnel Management, offered me the scholarship whose funding came from CIDA through the bilateral General Training Fund arrangement. Despite mid-way withdrawal of financial support, I had already set foot in it.

In the search for extra funds towards completion, Sylvia Osborn, Jim Mullin and Mike Bauer tried their best, in vain though, to have my CIDA funding reinstated. I found great support from the University through the offices of the President, Student Awards, Graduate Studies and the International Students. Dr Pedersen gave my wife and I a good listening, expressed understanding and referred us to Chuck Hill of the Student Awards where we got some bursary support. The Graduate Studies office, through Drs Steele and Peterson, was most useful in pointing us to other possibilities. Sandy of the International Students' Office was most understanding and managed to push my application for further bursary support. My department nominated me for the Special University Scholarship (SUS), an Ontario fee waiver, offered me Teaching Assistantship and, later, recommended me for the University Graduate Research Scholarship. The fees office, in turn, displayed patience with me by allowing the deferment of fees payment beyond set deadlines. Sandy Alfs and the Office staff in Department of Computer Science helped a lot in administrative matters. I owe plenty of gratitude to Sandy who, through her office, was instrumental in

I, otherwise, wouldn't have known about. In the often times of desperate financial quagmire, my brother Osumo Nyanchama generously stepped in to help. I am most grateful for his support.

This turned out to be what we call *Harambee* (let's pull together) in my country which became possible only with collective effort of these many goodwilling people. As the wise say in Kiswahili, *akufaaye kwa dhiki, ndiye rafiki* (a friend in need, is a friend indeed).

My wife, son and daughter bore the brunt of the difficulties encountered in the course of this study. Not only did I sacrifice their time and pleasure, but overcoming the stress occasioned by my long absence from home was not a mean task. I hope better times will come to justify these sacrifices. I sincerely hope that this experience and achievement will bring positive happenings to our lives in the future. For my son Kerara and daughter and "magokoro" (grandmother) Kemunto, I hope you will have an example from the persistence of your father in trying to achieve something for himself and the young family.

For my family (mother, father, brothers and sisters) in Kenya, I am most grateful for their understanding regarding my absence from home. Perhaps I would have helped more were I at home. But I hope my stay here would serve us all better in the time to come. My lessons are immense which I am sure will benefit all of us in the future when we unite under the Gusii skies in the land of our ancestors, the land of *Gutmwanda rigoti ri'egechure otaminyoke oikere.*

This appreciation would be incomplete without mentioning the many friends that gave me encouragement when the rough goings on appeared unbearable. The *kenya-net cyberspace* community of Kenyans and other East Africans created a home atmosphere and kept away potential loneliness in these distant foreign lands. Quite often, in times of stagnation in this work, this forum filled in the void and hence provided needed break from the monotony of this scientific study. I owe it to this Kenyan cyberspace family for helping forestall burnout.

This work I dedicate to my children Kerara Agata Matunda and Kemunto Nyakerario "grandma" Matunda. May you blossom like the flowers in the spring and live to accomplish what your heads prescribe as good for yourselves and your people.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION, GOALS & THESIS SCOPE

1.1 Introduction

Database security aims at preserving the *secrecy* and *integrity* of database information while preventing *denial of service*. Research in this area has focused mainly on government and military-like systems in an effort to emulate the paper world mainly in the military tradition. These research efforts have culminated in well documented models such as those of Bell and LaPadula [BL75] and Denning's information flow lattice model [Den76]. The United States Department of Defence's evaluation criteria for secure computer systems are set forth in the *orange book* [oD85] and its related network interpretation [oD87]. Others include the Government of Canada's product evaluation criteria [oC93].

Secrecy ensures that database information is available to only those authorized. It assures the **confidentiality** of the information. **Integrity**, on the other hand, ensures that system data are modified by trusted subjects and that the modifications are accurate and valid. Integrity assures the reliability of the information. Guarding against **denial of service** ensures that system data/information is available to authorized subjects when required. Excessively elaborate security mechanisms may result in denial of service. We do not address denial of service further in this thesis.

Despite the fact that results from military security can be applied to commercial environments (see for example Lee's mandatory categories [Lee88]), a distinction must be made between the different concerns of these two environments. While secrecy and integrity are emphasized in military environments, integrity is the overriding concern in commercial settings. Knowing military secrets such as attack plans can jeopardize a military mission just as altering the accuracy of this information. However, an employee's knowledge of a manager's salary may not be as serious as modifying the value of this salary: here integrity of the database is considered more important than the secrecy of the data it holds. The Clark and Wilson model [CW87] for commercial security recognizes these distinctions. It will be central to our study.

The other key concern in a commercial establishment is *separation of duty* [CW87, Tho91, San91, Kar88, Lee88]. It is common to have some task (say a cheque issuing process) subdivided into subtasks which are then carried out by different personnel. Separation of duty is enforced by ensuring that no single individual can perform more than (say) one of the subtasks that make up the task. For example, most establishments require that no one person can authorize both a voucher and the cheque based on that voucher. Barring collusion, separation of duty ensures that the rules specifying the manner of accomplishing a task are adhered to. Military security has no *explicit* provision for ensuring separation of duty. Despite the fact that it is possible (see for example Lee's [Lee88] realization using mandatory integrity categories) to specify constraints to enforce this principle, the task is not a straightforward one. For that reason, work such as [CW87, Tho91, San91, Lee88] and [Kar88] have resulted. Central to the Clark & Wilson Model (CWM) for commercial security [CW87] are the issues of integrity and separation of duty.

The interest shown in O-O database systems [AH90, ABD$^+$90, BCG$^+$87, Dit90, KC86, Kim90, LVV88, MS90, FKMT91, Osb89b, Osb89a, Cat94, LAC$^+$94] suggests that they have a future in modeling. O-O databases also have attractive features (types, classes, methods, inheritance, polymorphism, etc.) which we find useful for modeling complex applications. Of particular concern is the ability to specify the object interface and *apportion* it. From an authorization poin view, this enables us to base object information access on object interface partitions with different users/groups/etc. being authorized to different partitions. Since our focus is on database integrity, and given that database object transformations are based on methods (the object interface) in the O-O paradigm, our work associates authorizations to the object interface. This leads to object interface distribution among different users and different partitions facilitating access to the associated information. It is for these reasons that we propose an *object model* within whose framework we do our modeling.

The transactional nature in the commercial world is also an important component of our work. Indeed, the Clark & Wilson models suggests use of of *well-formed transactions* (WFTs) for information transformation. In our formulation, operations that perform database information transformation are constrained to be transactional in nature. Accordingly, we formulate a transaction model to provide a framework for such transactional transformations.

The *role* concept is implied in the Clark & Wilson model. A role is a collection of *privileges*, where a privilege facilitates access to database information. It is a unit of access rights administration. Once specified, a role is then authorized as a unit to users/groups/etc. Defining a role this way enables us to administer any number of privileges as a single unit for purposes of authorization. This makes the role concept central to this work. Hence we propose a generalized role modeling paradigm for role definition, organization and management. Starting from the basics of role relationships based on privilege sharing, we formulate the basis for role organization. The resulting structure is an *acyclic* graph with roles along a path having a *monotonically* increasing privilege set. We will demonstrate that this organization can simulate other structures such as lattices [RWK88, RWBK91], Ntrees [San88, San89], hierarchies [TDH92] and privilege graphs [Bal90].

Role-based protection brings several advantages (flexibility, least privilege, etc.) to access rights administration. However, without providing a similar level of security as traditional protection approaches, this approach to security would be unattractive. To demonstrate its strength with respect to traditional protection schemes, we explore role-based protection with respect to traditional protection approaches. Using information flow analysis techniques, we propose a means of determining the *consistency* of a given role-based scheme with respect to a security policy. This involves the use of subgraph isomorphism where a given role definition scheme is consistent with (and hence secure with respect to) a given security policy, if the graph of its information flows is a subgraph of the graph of information flows due to the security policy. With the use of matrix analysis the performance of this process is $O(n^2)$, where n is the number of roles. Further, using similar information flow analysis techniques, we demonstrate that we can emulate mandatory access control using role-based security. This involves imposing conditions that realize secrecy and acyclic information flows on a given role scheme.

Using the proposed object model, transaction model and role organizing framework, we propose a formulation of the Clark and Wilson model. In particular, we show how to model concepts of constrained data items (CDIs), transformation procedures (TPs), well-formed transactions (WFTs), integrity verification procedures (IVPs) and separation of duty.

The rest of this chapter provides further insight into this work. In section 1.2 we discuss the motivation for this study. Section 1.3 offers the scope of this work. We discuss the organization of this thesis in section 1.4 and offer a summary of what, in our view, is the contribution of this study in section 1.5.

1.2 Motivation

This work aims at exploring security issues in non-military/non-government applications where integrity is the key security requirement. The Clark and Wilson model [CW87] offers requirements that are suitable for application in such environments. Such environments emphasize integrity more than they do secrecy. Indeed, a key difference between the two application environments is the difference in emphasis placed on the two. As more and more commercial organizations computerize, it is our view that greater emphasis will be placed on information integrity as opposed to secrecy.

A second motivation arises from the observation that roles offer a "natural" approach to distribute and administer responsibilities and obligations in organizations. It is our view that we shall see greater tendencies in this direction given the search for different paradigms of modeling the administration of access rights in organizations. Consequently, we are driven to search for ways of defining, organizing and administering roles. Further, given the central nature that the issue of security plays in the real world, it is only appropriate to study the implications of role-based protection. As such, we study role-based security in the light of information flow approaches and mandatory access control.

A third consideration is that the O-O paradigm offers a semantically rich approach to modeling real world entities. Since organizations and their data, like any real world entities, are likely to be complex, models such as the relational database model are not likely to be expressive enough to capture this complexity. It is our view that O-O systems provide such expressive power for specification and manipulation of such complex entities. Our work will exploit this inherent expressive power of the O-O paradigm.

1.3 Thesis Goals & Scope

The following offers a summary of the goals we intend to achieve and hence the scope of this thesis:

1. The term role has been used extensively in the literature (see for example [CW87, DM89, San91, Tho91, KM92] and others). Invariably, its meaning depends on the context in which it is applied, prompting Baldwin [Bal90] to opt for the term *named protection domain* (NPD) instead. The problem lies with the lack of a formal definition for the concept. Hence in order to use the term role meaningfully, we need a formal definition for it. Part of this work focuses on realizing a formal definition for the role concept. In doing so, we view a role as a unit control that helps realize some system functionality. It can be seen as a collections of functions that help achieve some duty requirements. Moreover, these functions themselves must be formally defined. This work aims at formalizing the functions that constitute a role and hence yield a formal definition for the term role.

2. Role-based protection is implied in the Clark and Wilson model [CW87]. Although aimed at meeting security and integrity requirements in commercial environments, role-based protection can be applied in various environments other than those proposed by in the model (see for example Ting et al. [TDH92]). Thus the second goal of this thesis is to explore some theoretical properties of role-based protection as it relates to known security principles such as *information flow* and *access control*. By viewing roles as providing access to some information context, and considering any implicit information flows due to role definitions, one can determine whether a given role definition scheme is consistent with a given information flow policy. Another interesting tangent of this exploration is the examination of what access control schemes are suitable for role-based protection and any theoretical properties they may have. An obvious investigation is how, for instance, mandatory access control could be achieved via role protection strategies.

3. Role organization is also an important component of role management. There are various role organizing structures such as lattices [RWK88, RWBK91], hierarchies [TDH92], Ntrees [San88, San89], privilege graphs [Bal90], etc. We intend to provide a generalized framework for role organization and management which captures the key properties of these role organization structures and facilitates the ease of administration of access rights in a system. By investigating these key properties, we crystallize them for use in our formulation. The proposed role organization model is a graph where the nodes and edges of the

graph are roles and privilege relationships between the roles, respectively. Role relationships are derived from privilege sharing. Further, we shall propose role administration algorithms that preserve the role organization properties. We shall demonstrate that the model simulates other role organization structures. This forms our third goal of this work.

4. Given the advantages that roles bring to protection and those due to the O-O approach to modeling, our fourth goal is to exploit the advantages of this combination to formulate a protection scheme. In an O-O system, operations on objects determine object information manipulation. Hence it makes sense to offer information protection at the operations level. By associating methods with the role privileges, we are able to offer segmented access to object information. A role can be seen as a collection of interface segments of some objects.

5. Given our aims are to offer a role-based O-O model for commercial security, our fifth goal is to demonstrate the realization of properties of the Clark and Wilson model. To do so, we define database objects (at least those whose integrity must be maintained) as constrained data items (CDIs) and operations (methods) as Well-Formed Transactions (WFTs)[1]. Our approach is to associate with every execution process some assertions which must be satisfied before execution can take place. Since our formulation is based on O-O pr'nciples, we modify associated method executions with assertions about their executions. This imposes WFT requirement on executions. The transactional execution requirement is governed by a transaction model.

   Separation of duty is a major requirement of the Clark and Wilson model. Enforcing separation of duty requires audit information pertaining to the object being accessed. Thus it is important to investigate how to capture audit information pertaining to a given object. The object history concept provides a way out. It requires that an O-O model provide a means of capturing object history within the object state.

[1]Both Transformation Procedures (TPs) and Integrity Verification Procedures (IVPs) are termed WFTs.

1.4 Thesis Organization

This section offers a brief outline of the organization of the body of this thesis. References to the appendices will be made if and when necessary.

1.4.1 Chapter Two

In chapter 2 we offer an overview of commercial database security [CW87] and distinguish it from government/military security. The fundamental distinction is the difference in emphasis placed upon secrecy and integrity. In military type security confidentiality and integrity of information are paramount. In modeling security, e.g. using the Bell & LaPadula model [BL75] the rules governing information access results in a conflict between secrecy and integrity. For instance subjects can write information that they cannot view (i.e. information of higher clearance than subjects' clearances). This leads to *polyinstantiation* which is a major research issue in computer security (see for example [JS90, JS91a, LH91, SW92, CY92b]).

In commercial security in general, and with the Clark & Wilson model in particular, the integrity of information is given higher priority than its secrecy. Thus the correctness (accuracy) of transformation procedures, the correctness of the transformations themselves and the validity of the state of database objects (the information bearing entities) are given emphasis. Hence all transformations on object state must ensure that they take in data in a consistent state and transform it into another consistent state. They must also ensure that any inconsistent input into a transformation procedure, if acted upon, would result in a consistent output state; otherwise the transformation will be rejected. Yet another important requirement is maintaining the correctness of the transformation such that it is well protected from tampering. All these provisions are aimed at preserving information integrity.

As mentioned in section 1.1, separation of duty is a component requirement of commercial security as espoused in the Clark and Wilson model. The concept aims at distributing the responsibility, authority and obligation across different individuals in a system. Thus a subject responsible for approving a voucher should not approve the associated cheque; where two signatures are required, no single individual should execute both. Accordingly, the model requires that we not only keep track of individual authorizations but also object history [Kar88, NO93a].

As well, authorization assignments must be such that it is possible to associate subjects with transformation procedures, transformation procedures with data items and ensure that these relationships are not violated. Authorization to a data item then depends on this authorization information as well as object history, commonly known as *audit information*. What we present in chapter 2 is a summary of the Clark & Wilson model. For more details the interested reader is referred to [CW87, Kar88, Lee88] which offer extended expositions.

1.4.2 Chapter Three

Our intention is to utilize O-O concepts for commercial integrity modeling. This is the subject of chapter 3. The beauty of the O-O paradigm is that its concepts map almost naturally to our intended task. For instance, with the *class* concept we can define specific object types with specific structure and behaviour. Methods, being the only means of object state manipulation, provide a means of segmenting the object interface. These partitions can then be authorized to different users/groups/etc. based on need. Inheritance facilitates incremental reuse of behaviour and structure (both definitions and implementations). Others concepts like polymorphism are also suitable in that we can send the same message to different objects with the responses depending on the message receiver.

Accordingly, we propose an object model in chapter 3 which has many concepts in common with others such as [AH90, ABD+90, BCG+87, Dit90, Kim90, LVV88, MS90, FKMT91, Osb89a, Cat94, LAC+94]. We recognize that the task of espousing a whole model in the sense of O-O models, would itself be onerous. Indeed, with controversies still raging on what should constitute a model and how certain concepts should be modeled, we feel that such a task would be beyond the scope of this work. Consequently, what we present in this chapter are some skeletal aspects which have a bearing on our modeling. They can be viewed as concepts which can be specialized for and built into any O-O model.

Unlike these models we do not propose a query language for the reason that all accesses to database objects is via transactions. Hence our model takes the concept of method access to objects and clothes it in a transactional paradigm. This choice is based on ıeal life observations in the commercial world where access to objects (such as cheques, vouchers, audit trail, etc.) are transactional in nature as required by commercial security. We impose a restriction that all object transformations must be transactional. Thus methods must themselves be transactional or act within some transactional operation.

We define object interface partition as a subset of the methods associated with an object. Authorization to object information is associated with authorization to some interface partition. Making all transformations transactional means that object interface partitions are "wrapped" in some transactional executions.

An important extension in this model is the addition of the notion of *object history*. Object history (or audit information) is important for the enforcement of separation of duty. In our model, this history is a component of the object state, where the object state is determined by the values of its attributes. An object state can thus be partitioned into history and non-history components, *H_state* and *NH_state*, respectively. The former tracks required audit information while the latter keeps the rest of the object state. Along with this concept of *H_state* is the update constraint that requires that the object history be updated every time the object is accessed or when there is an attempt at such access. Updating the *NH_state*, on the other hand, depends on whether or not the associated transaction commits or aborts. The updated *NH_state* takes effect only on commit.

The O-O model proposed in chapter 3 formally presents the concept of object history and demonstrates its incorporation into O-O modeling. History is defined as a series of events where an event, as a unit of object history, captures the nature of audit information required for separation duty. This generalized definition of the term event is attractive as it allows for specialization to suit a given application.

1.4.3 Chapter Four

The Clark and Wilson model requires operations to be well-formed transactions. Therefore, operations in our envisioned application environment must be transactional. Consequently, we propose a transaction model in chapter 4 for this purpose. This formulation builds on the traditional transaction model properties: atomicity, consistency, isolation and durability. Unlike the traditional model, we allow nesting where a transaction can have several child transactions and propose rules governing commit/abort actions for parent/child transactions. Another important feature of this model, the *compensating transaction*, is intended to cater for recovery. For every transaction defined, we need a compensating transaction to undo its effects in case of a failure or an abort.

The *script* [WR92] is yet another feature of our transaction model. A script is a specification of the order of execution of one or more transactions. Scripts use serial, parallel and iterative operators to prescribe a desired order. It is shown that any ordering of transaction executions using these operators realizes a transaction itself. Scripts are designed with particular results in mind which are determined by the specified order in which its transactions execute.

1.4.4 Chapter Five

An important requirement in any organization with many people accessing shared information, yet with differing privileges, is some organizing framework that would reduce the burden of specifying and managing authorizations while assuring information security. In *multilevel security,* MLS (which implements mandatory access control, MAC), information is classified while users are assigned clearances similar to the information classes. The task of authorization is thus reduced to that of determining what clearance should be given to what users. Administration of authorization is reduced to the task of ensuring that users (subjects or processes acting on their behalf) are cleared for the information they access. Ensuring security becomes the task of making sure that users access information according to the simple security and *-properties of the Bell and LaPadula [BL75] model according to their classification/clearance attributes. As well, information security guards against tampering with authorization information. In discretionary access control (DAC), users have the authority to determine what other users can access information under their *ownership.* There exist schemes such as *owner, group, other* (for operating systems) that also group users into user groups. Access to information depends on the access rights of the group(s) a given user belongs to.

In commercial security and with the Clark & Wilson model in particular, the concept of roles is implied. This is the subject of chapter 5. A role is a collection of privileges where a privilege determines an authorized user's access to the associated object. Besides acting as privilege (access rights, obligations, etc.) holders, roles have relationships with other roles. In a nutshell, in our formulation, a role captures both the *function* [DM89] (actions, rights, obligations, etc.) and *structure* [DM89] (role composition and relationships with other roles) in a system. A role is authorized to a user/group as a *single unit* and allows such a user/group to exercise the privileges specified by the role. In role-based protection, the tasks include specifying what roles exist, what privileges constitute a role and which users are authorized to the roles. Once specified, determination of user access to information reduces to ensuring that a user is authorized to the role and that the privilege being executed exists in the role's privilege set.

The advantage with roles is the *flexibility* they bring to access rights administration. This flexibity is mainly due to the fact that a role exists separately from both the user and the resource (object, etc.) accessed via the role. User and resource relationships lie outside the role function and structure. The challenge for an authorization scheme designer is to maintain desired role structure and functionality, maintain user relationships (if any) as well as maintain resource relationships (if any). "Disturbing" any of the specified relationships is a security violation.

Role-based protection can accommodates wide-ranging security policies. While our intention is to realize the Clark and Wilson model concepts, an interesting study focuses on the application of traditional security concepts to roles. Thus information flow analysis techniques are used for the analysis of secure role-based protection designs. This involves the comparison of information flows inferred from both a given policy and a given design (or implementation). A role definition scheme is consistent, hence secure, with respect to a given policy if the graph of its information flows is a subgraph of the graph of information flows due to the security policy.

Another important tangent is to determine whether we can realize mandatory access control using role-based protection schemes. In doing so we impose both the secrecy requirement in authorization and acyclicity of information flows. Thus we demonstrate that we can achieve a level of protection equivalent to what is achievable via mandatory access control.

1.4.5 Chapter Six

Once roles are defined, there is a need for a role organizing framework to facilitate their administration and management. Chapter 6 presents such an organizing framework for roles. By exploring the the basic relationship among roles, we extract the key relationships that form the foundations for this organization. These include *partial, common* and *augmented* privilege sharing. **Partial** privilege sharing refers to a *junior-senior* relationship between roles in which the privilege set of the junior role is a subset of that of the senior role. **Common** privilege sharing is the case where two roles have the same junior role while **augmented** privilege sharing refers the case where two roles have a common superior. Other important concepts are those of *minimum* and *maximum* privileges. These enable us to ease the administration of role access rights.

The result is an *acyclic* structure for role organization based on privilege sharing with a *monotonically* increasing privilege relationship for roles in a given path.

The model is intended to ease access rights administration in a system. To that effect it maintains key properties such as *acyclicity* and role uniqueness based on role privilege sets. To facilitate role manipulation, algorithms are given that ensure the retention of the model properties.

Various structures have been proposed for role organization. These include hierarchies [TDH92], Ntrees [San88, San89], Domain Definition Tables (DTTs) [Tho91], Lattices [RWK88, RWBK91] or privilege graphs [Bal90]. Whatever the form, what results is a structure that defines both the role function and structure. Indeed, except for DDTs, these structures capture both these role specifications. The overall structure, which we term the *authority structure*, determines the distribution and exercise of authority in the system.

We show that our formulation can simulate these other structures. Hence we claim that it offers a more generalized form for role organization than those mentioned above. The proposed structure is formally defined using various operators with their appropriate semantics. Since the operators specify role relationships, they, in effect, determine the distribution and sharing of syster.. privileges.

1.4.6 Chapter Seven

Chapter 7 utilizes the object, transaction and role models of chapters 3, 4 and 5 to formulate the properties of the Clark and Wilson model. In the ob:ect model, object manipulations are transactional. These are authorized to the roles hence effectively distributing the *object interface* among roles. Since object access is transactional, privileges are defined as transactions. Role relationships (hence role structure) is capture.l by the role graph model. Defining privileges as well-formed transactions and objects as constrained data items, we show how the Clark & Wilson model properties are satisfied. As well, we demonstrate how to enforce separation of duty within the role graph model.

This chapter also addresses the issue of *conflict of interest.* The most common conflict is when a user is authorized both the administrative and execution privileges of a given role. Imposing a conflicting relationship on a given set of roles partitions roles into groups. We show that such a relationship partitions a role organization graph into independent roles.

1.4.7 Chapter Eight

In the last chapter we recap the work covered in this thesis. Here we offer a summary of our work and what, in our view, constitute the key contributions and achievements of this work to research in computer security.

1.5 Summary & Thesis Contribution

In this chapter we have outlined the motivations for this work which include the recognition of the advantage of utilizing role modeling and adapting it for role protection. Moreover, given that our key concerns are with information protection, understanding the use of role-based protection is important. As well, the choice of commercial security criteria is based on the observation that applications, other than military ones, will consider such concerns paramount. The choice of the O-O approach is due to its rich semantics that eases the task of data modeling.

We also outlined the goals of this thesis. Among them: exploring and developing theoretical notions for role-based security, studying role-based protection in the light of traditional security approaches (information flow and mandatory access control), modeling role organization, application of the role organization in modeling commercial security. A further goal is to use O-O principles to specify and manage access to objects in our model.

We have also outlined the scope of this which includes studying role-based protection approaches, marrying role-based protection with O-O principles and formulating a means of satisfying principles of the Clark and Wilson model.

We regard the following as the major contributions of this work:

1. **Formalizing the Role Concept**

   This study explore the basis of role-based protection and formalizes the the role concept. Starting with a formal definition of privilege, it is possible to define a role as a collection of privileges. At the beginning of this study, we could not find a formal definition of the term role. Indeed, some, notably [Bal90], eschew the use of the term role arguing that it means different things to different people. Our work shows that we can narrow this spread of meaning through formalizing the concept.

2. **Modeling Role Organization and Administration:**

Role administration, and by extension access rights administration, can be a very complex process in role-based security systems. The desire to have a model to ease this task requires the development a model and associated algorithms for role-administration. Using basic role relationships, this model realizes a formal framework for role administration. Role relationships themselves depend on shared privileges. Analyzing these relationships forms the basis for derivation of the role organization model. To demonstrate the expressive power of our formulation, we show how to simulate other role organizing frameworks.

Suggested operations on the model are meant to preserve model properties. The proposed operations facilitate tasks such as role addition, deletion and partition. These operations which are intended to ease access rights administration are presented in the form of algorithms.

3. **Roles, Information Flow & Mandatory Access Control:**

   The question of information flow in role-based protection is studied as well. In particular, we study the application of information flow analysis principles to the definition of role-based protection schemes. Taking a system information flow policy and considering a role scheme as an implementation, we propose a methodology for ensuring that a given implementation does not violate system security policy. Using graph theory concepts we demonstrate how to represent policy and implementation information flows as two graphs. Applying the concepts of *subgraph isomorphism*, we define implementation consistency to be the case where an implementation information flow graph is a subgraph of the policy information flow graph. Consistency, thus, can be used as the criterion for security of a given scheme with respect to a given policy.

   With respect to mandatory access control (MAC) our study focuses on demonstrating that MAC-like protection can be realized using role-based protection schemes. Our analysis follows from the application of information flow principles. Imposing the restriction that information flow must be unidirectional and acyclic, we can realize similar protection with roles as can be achieved by *multilevel security.*

4. **Realizing Principles of Clark & Wilson Model:**

   Using the role concept and O-O approaches we demonstrate the realization of the principles of the Clark and Wilson model for commercial security. In particular, we introduce the concept of interface partition and distribution among roles. Object histories are also introduced for use in enforcing separation of duty. By defining the interfaces as WFTs and the objects as the CDIs, we guarantee the realization of principles of the Clark and Wilson model.

5. **Using O-O Principles To Track Audit Information:**

   We propose an object model which incorporates commercial integrity requirements. We thus have objects which track their histories, object histories that are updated whenever the object is accessed, object method executions dependent on the object history, etc. While we demonstrate that the histories can be modeled as object attributes, this is not the only manner of keeping track of such histories. The important thing is the abstract concept of object history and the requirement to keep track of it as part of the object state.

6. **Script-Based Nested Transaction Model:**

   We also propose a script-based nested transaction model. The script prescribes the order of execution of operation in a given type/class of objects. Each such script has a compensating script which compensates for the actions of the script. We allow for arbitrary commit of children transactions but consider a transaction committed when the parent and all its children have committed.

As a matter of information we observe that many of the concepts that this work addresses are not new. For example the role concept has been around for a while as has been others like the O-O approaches, information flow analysis and its application to security. However, this work brings together these known ideas and weaves them together in a manner that realizes the requirements of the Clark and Wilson model for security. This then, in a way, summarizes the key contributions of our work.

An a matter of information, parts of this work have been published variously as references [NO93a, NO93b, NO94c, Nya93a, Nya93b] and [NO94b]. The PhD research proposal [Nya93b] formed the basis and set the scope of this work.

CHAPTER 2

COMMERCIAL DATABASE SECURITY

2.1 Introduction

Computer security, in general, is concerned with *unauthorized* users. It is intended to safeguard the *secrecy* and *integrity* of information while guarding against *denial of service*. Database security focuses on database information and aims to ensure the *secrecy* and *integrity* of the data and guard against *denial of service* for authorized access to the data.

Secrecy aims at keeping information *confidential* while making it available to *authorized* users. **Integrity** aims at keeping information accurate and consistent by guarding against unauthorized modifications (accidental or otherwise) and ensuring the correctness of the information. Preventing **denial of service**, on the other hand, aims at ensuring that authorized users have access to information when and in the form required.

Depending on the application, different emphasis can be placed on secrecy and integrity. For instance in military security, secrecy is just as important as is integrity. Unlike military systems, commercial environments emphasize information integrity more than its secrecy. The risk associated with information modification (e.g. a bank account balance) is much higher than that associated with knowledge of (say) the value of the same bank balance. In a nutshell, secrecy is not as critical a consideration in such a case as it would otherwise be in a military situation. However, this is not to say that commercial systems do not require secrecy (e.g. proprietary information is usually critical to a company's success) but to drive home the point that in some applications, integrity may be given greater consideration that secrecy or vice versa. This distinction in the emphasis of security requirements for military and commercial environments has led to the terms: "military" and "commercial" security. In the former, both secrecy and integrity are just as important. For instance revelation of secret information pertaining to some project can place the project in as much jeopardy as the modification of the information. In the latter, the overriding concern is the integrity of the information. Hence key concerns revolve around the validity of the data and the accuracy of the transformations which act on the data. Additional concerns include conflict of interest and separation of duty. The distinction between the latter terms will be clear as we proceed.

This chapter presents an overview of the differences between secrecy and integrity, summarizes commercial security requirements as propounded by Clark and Wilson [CW87] and offers a summary of related work. In addressing related work, references are made to roles, which are implied in the Clark and Wilson model, and role organization. The distinction between secrecy and integrity is given in section 2.2. In section 2.3 we discuss the differences between approaches in military and commercial environments. This distinction is important for it forms the basis for enunciation of the Clark and Wilson model which is the subject of section 2.4. The key issues discussed here focus on the basic principles underlying the model and the properties a system must satisfy in order to meet said commercial security requirements. This section offers only an overview of the model. Detailed aspects of commercial security can be found in [CW87, Tho91, San91, Lee88, Kar88]. In section 2.5 we briefly outline some related work. A more detailed brief is given in appendix A. Section 2.6 is a summary of the discussion covered in this chapter.

2.2 Secrecy & Integrity: A Distinction

Database security distinguishes subjects and objects. Subjects are the active entities that manipulate database information. Subjects can be users and/or processes acting on their behalf. Objects are the information bearing receptacles.

A central concern of database security is the *secrecy* of the database information. Secrecy is the ability to keep database information *confidential.* It is the ability to ensure that information is available only to authorized subjects. Violation of confidentiality is a security breach. Such violation can take various forms. It can occur via direct leakage, where a system security's retinue may be weak. It can be indirect leakage via *Trojan horse* attacks, inference or *covert* channels. Covert channels exist where information inadvertently leaks either via storage or timing. The former involve leakage of information into some storage channel as in the case where a process, executing legally, inadvertently writes information into a storage device such as a file. The latter, on the other hand, pertains to signalling via process behaviour; observation of such behaviour effectively leaks information. Inference, on the other hand, involves the *aggregation* of "legally" accessible facts to deduce unauthorized facts. Aggregation and inferences are major study issues in database security research (see for example [Hin88, Lun89, Lin91, Wis90, MHT92]).

Another important database security concern is integrity of database information. It aims at preventing inadvertent modification of database objects, accidental or otherwise. It ensures that authorized subjects transform data and that such modifications leave the information in a valid state. Essentially, database integrity deals with authorized modifications and the accuracy of the modifications. Accurate modifications ensure that database information is valid. Preserving the integrity of information, in turn, assures information *reliability*. To be reliable, the information must be accurate, hence correct. Preserving integrity involves the correct and legal modification of the information. This can be done by ensuring that the subjects authorized to carry out modifications do so correctly. Such correctness is set out in the security policy from which the correctness criteria are derived.

Secrecy and integrity are different, although both are desirable for database protection. While secrecy is concerned with keeping information confidential, integrity is concerned with ensuring that such information is not only accurate, but also correct and valid, where correctness and validity are specified based on defined criteria. While secrecy deals with unauthorized access, integrity is concerned with both unauthorized access (such as masquerades, accidents, etc.) and authorized access (such inconsistent modifications). In the latter case, integrity is concerned with the accuracy and correctness of database information by ensuring that any modifications meet the system integrity constraints. Generally, this form of integrity is referred to as *consistency integrity* [SD87]. As Date [Dat83] demonstrates, it is concerned with the concepts of *accuracy, correctness* and *validity* of database information and the operations on the information [Dat83]. System integrity requirements ensure that the data are not exposed to accidental and/or malicious modifications and/or destruction. Our work focuses on both authorized and unauthorized access. We are concerned with the integrity of transformations as well as the ability to forestall unauthorized modifications.

Other forms of integrity exist including data integrity, entity integrity, label integrity, system integrity, referential integrity, etc. To ensure database integrity in trusted systems, the DBMS must enforce integrity control by: checking whether given modifications are authorized and whether such modifications result in information that is consistent and correct (based on some criteria). One such application of integrity control is in *concurrency control* where many transactions can have concurrent access to shared data. When the effect of such transactions is to modify the data, a correctness criterion is required to ensure that the database remains in a consistent state after completion of executions. In most applications, the serializability criterion is used as the correctness criterion in situations of concurrency control.

It is not easy to meet the requirements of both secrecy and integrity. This then leads to conflict due to the need to satisfy both of these requirements. For example in the Bell and LaPadula model, which enforces multilevel security, subjects can write to levels equal to or higher than their clearance while they can read information at levels equal to or lower than their clearance. Consequently, subjects can create/modify information that they cannot view which leads to the possibility of creating objects with the same names as objects already in existence. The result is that there can be many objects with the same name(s) but created by subjects with different security clearances and which may have differennt classifications; this is termed polyinstantiation. In order to reconcile and determine which of the multiple instantiations should be utilized requires intervention outside the model mechanisms. Polyinstantiation remains a major research concern in computer security [LSS+88, JS90, JS91a, JS91b, LH91, CY92a, SW92].

2.3 Why Commercial Security?

Integrity and secrecy carry similar weight in government and military environments. Violation of confidentiality may be as much of security violation as is violation of integrity. The knowledge of a certain piece of information may be as serious as the modification of the same or another piece of information.

The same cannot be said for a commercial environment where knowledge of a certain piece of information may not be as disastrous as modification of the same or a different piece. Emphasis here tends to weigh on the integrity (hence reliability) of information in contrast with the emphasis in a military setting. As an example, a company may publish the salaries of its executives and may make it accessible to all employees while barring alteration of this figure to all other employees except some. Integrity is of higher concern than confidentiality in the commercial world.

These distinctions require the investigation of environment-sp⌐¬ific concerns to build models that address these concerns. While models like that of Bell & La-Padula [BL75] and Denning's lattice model [Den76] may be tailored to meet certain requirements in commercial applications (for example, see Lee's [Lee88] mandatory categories), they fail to do so in some circumstances (see [NP90]). Traditional models are thus unsuitable for modeling commercial security requirements. An example of such a drawback relates to *multilevel security* (*MLS*) that is common in military type security models. Here, there are partially ordered security levels (as in Denning's lattice model [Den76]) and the security mechanisms prevent the flow of information from higher to lower security levels. Such a scheme has no "natural" mapping to specifications in commercial applications where there can be multidirectional information flows.

To formulate and enforce commercial security requires a specification of roles required for the processing environment. A role will be equipped with sufficient "capabilities" to meet duty requirements. Having defined roles, users are authorized to act in specified roles. Depending on the requirements of a given application, a user can be authorized different roles. For instance, a manager can perform the roles of both manager and clerk. For ease of management, there can be specified a role organization structure for the administration of roles. For example, they can be ordered to yield a role lattice [RWK88, RWBK91], hierarchy [TDH92] or some other suitable structure with specific rules governing the role relationships. In general, the rules governing information processing will specify and regulate the relationships between the different roles. As well, they govern the relationships between individuals and defined roles. Application of these rules can be mandatory in that they are system-defined and cannot be altered in the course of processing, nor can they be modified by the individuals to whom they apply.

In military situations, *clearance* is central to information access and manipulation. Once users have a certain clearance, the rules on what they can do are clear, e.g. the Bell and LaPadula model the no write-up and no read-down rules [BL75] govern access. A *need-to-know* requirement may also be imposed to limit subjects accessing all information at any one security level. A combination of mandatory and discretionary access control realizes this [oD85].

These reasons provide the motivating force behind Clark & Wilson's model (CWM) [CW87]. Its emphasis, as said in the foregoing paragraphs, lies in commercial security concerns as opposed to those of the military. Clark & Wilson contend that in a commercial environment, integrity is of greater concern than is secrecy. Indeed, the accuracy and correctness of mechanisms in a commercial environment override concerns for revelation of information.

2.4 The Clark & Wilson Model

Our work addresses the concerns of integrity in commercial environments. In particular, we focus on security requirements as specified in the Clark and Wilson model [CW87]. This section will be an overview of the basic principles of the model which form the basis for commercial security.

To achieve the desirable commercial security, the model enunciates an idea based on two properties, viz: Well-formed Transactions (WFTs) and Separation of duty. A WFT can be seen as a programme which assures the integrity of the data it manipulates [Tho91]. Separation of duty, on the other hand requires that different users execute different subtasks of a given major task. We will explain these in turn in the following sections. The model properties which assure these key principles will be addressed as well.

The enforcement of the model policies is regarded as mandatory: given users' access to information is based on user authorization to execute system-defined operations. Further, user access rights cannot be transferred to third parties. Moreover, users cannot exercise administrative rights over their own access rights. As in traditional MAC, this case requires that subjects' and objects' attributes are the only basis for granting authorization according to specified rules.

In general, the Clark and Wilson model can be seen as specifying system behaviour *a priori*. This behaviour is determined by the operations defined on the data objects. User-operation relationships govern the execution of the operations and hence the exhibition of the specified behaviour. Clearly, this makes the model static.

2.4.1 Well-Formed Transactions

A well-formed transaction (WFT) is a program that has been certified to perform a specific operation that preserves or introduces integrity policies [Tho91]. A WFT constrains a user from using accessible data in ways other than those specified. Presumably these specified ways must have been certified to preserve and/or ensure the integrity of the data. WFTs are suitable for the preservation of data integrity in cases that would otherwise be difficult to handle with a generalized security policy.

They can also be used to ensure that a given piece of code contains no *Trojan horse* [Tho91]. They can be certified to be correct, a task whose extensiveness and expense would depend on how stringent the security requirements are.

A WFT also ensures that the database is consistent once the WFT execution is over. This happens in that each WFT execution starts with a correct and consistent database state. Given that it must execute correctly, then it must, at the end of the execution, leave the database in a correct state. Each transaction involves a certification procedure that ascertains that the database is in a consistent state before start of execution. Should the certification step fail, the transaction will abort.

In the formal Clark & Wilson model, WFTs are either Integrity Verification Procedures (IVPs) or Transformation Procedures (TPs). Both act on database objects referred to as Constrained Data Items (CDIs).

There is key difference between TPs and IVPs [AAL+93]. TPs act on database objects transforming their state and hence changing the database state. They are the main agents of database information transformation in the model. TPs transform CDI states while assuring the validity of the objects. They are designed to take a correct input CDI state to another correct output CD state. IVPs, on the other hand, ensure the integrity of both the data and the effects of the TPs. IVPs can be seen as confirming that the integrity of CDIs is maintained. They act more like "auditing" procedures that assure CDI integrity. In other words, even when the TPs' effects may be correct based on the definition of state validity, their integrity may be compromised. For example, in double-entry book keeping, a debit entry in one place must be accompanied by a credit entry in another. A double-entry IVP procedure would ensure that TPs carrying such a double-entry process actually honours the credit-debit book entry requirement. This audit program will be designed in a manner that ensures that the accounting books balance.

CDIs are the data items whose integrity the system must maintain. These must be identified and labeled as such. Desired integrity polices are then enforced via IVPs and TPs. IVPs confirm that all CDIs conform to the integrity specification whenever an IVP is executed. On the other hand, TPs enforce the WFT concept and are responsible for transforming CDIs from one valid state to another valid state. IVPs and TPs must themselves be certified with respect to some integrity policy. Such certification is the responsibility of the system security function (e.g. the system security officer, SSO, or some system security privileges authorized such an SSO) and could be done either manually or automatically. This certification and proper enforcement of the integrity policy by the system assures system integrity.

2.4.2 Separation of Duty

Separation of duty is applied where several people are required to perform a given task. Such a task may then be broken down into subparts and different people assigned these different subtasks. This ensures that the responsibility associated with the performance of a given task is distributed among different individuals. It is aimed at forestalling fraud, barring any chance of collusion. It is applied where several individuals (or processes acting on their behalf) are required to perform a given task. The task is then broken into subparts which are assigned to different people. Every individual is then required to perform (say) only one of the subtasks with the restriction that none of the individuals can perform more than a given number or all of the subtasks. An example is the check-voucher process where a voucher initiator cannot authorize the writing of a cheque nor sign the cheque.

In specifying separation of duty, the Clark and Wilson model requires that the system maintain user-TP and TP-CDI relations. From these can be derived user-CDI relations which specify user authorization to manipulate CDIs in the manner specified in the TP-CDI relationship. Consequently, in executing the subtasks of a given processing of a CDI, we can keep track of the user identities that have been involved in the execution in order to enforce separation of duty. Therefore, separation of duty requires that we track CDI execution history.

Using the notion of roles, it is possible to realize a separation of duty system that assures that an individual performs roles as specified in the rules governing them. Such rules may include the requirement that an individual cannot perform more than one role in the processing of some task. The Clark & Wilson model is suitable for roles. In the cheque-voucher example, different subtasks may be assigned different roles in the system. Ensuring that no single person performs more than one of the assigned roles ensures that no single person can initiate, authorize and sign a cheque.

2.4.3 Model Properties

To accomplish the foregoing, the model lays out desirable properties of the system that would assure integrity. These properties fall into two categories: *certification* and *enforcement.*

Certification properties serve to assure that before any executions commence, all the constraints pertaining to such executions are met. These include whether or not all the data is in a valid state, whether or not the procedures that operate on these data are valid, whether or not the procedures produce valid transformations on the data, etc. Among the certification properties in the CWM model are [CW87]:

1. IVPs must ensure that all CDIs are in a valid state whenever an IVP is run. This is similar to validating the state of instantiated input parameters. Input parameters assume values from certain domains. The domains themselves have constraints that must be met by the instantiated parameters. The validation process ensures that the parameters belong to their appropriate domains and their values meet the constraints specified for the domains.

2. certification that all TPs are valid and that any such transformation must take the input CDIs to a valid final state. The system (or more appropriately the SSO) must specify the relation between a TP and the CDIs it takes as input. The task is to verify that the TPs execute correctly, i.e. they meet their specifications. In other words, given a correct input state of a CDI, the CDI will be in a valid final state after TP execution. For the output state of TPIs to be valid, the relationship between TPs and CDIs must hold.

3. all relations between a user and a TP must must be certified to ensure separation of duty as specified in the desired policy.

4. All TPs must be certified to write to an append only CDI (e.g. an audit log) all necessary information to enable reconstruction of the operation. This is of particular importance in commercial establishments where a high level of accountability is required. Without the ability to reconstruct the actions of TPs, retracing the operation sequence will be a difficult (if not an impossible) task.

5. Any TP that takes an unconstrained data item (UDI) must be certified to perform only valid transformations for any possible value of the UDI. Such a transformation must take the UDI input and transform it into a CDI or reject the operation. This helps prevent valid manipulations of UDIs that yield UDIs hence invalidating the whole operation.

On the other hand, the enforcement properties assure the maintenance of relationships between TPs and CDIs. These include [CW87]:

1. The system must maintain the specified list of relations specified between TPs and CDIs and ensure that CDIs are manipulated only by TPs according to these relations. Any TP operation on a CDI must be according to some specified relation. Hence there must be no operation on a CDI except by a TP and according to some relation.

2. The system must maintain a list of relations between system users and TPs of the form: **(usid,(TP$_i$(CDI$_a$, CDI$_b$, $\cdots$))** that associate a system user and a TP in the system. It must ensure that the only executions that occur are those described in the relations.

3. A user attempting to execute a TP must be authenticated before execution proceeds. Such authentication must include the user identity as well as the authorization to execute the given TP.

4. Only agents with permission to certify entities may change the list of such entities associated with other entities and specifically associated with a TP. Entities with certification permission on an entity may not have execute rights on the same entity. This imposes separation in that entities with execution rights on an another entity are not at the same time responsible for determining these rights.

To meet the requirements of commercial integrity as specified in the Clark and Wilson model, a system must meet the certification and enforcement requirements.

2.5 Related Work

The Clark & Wilson model, the basis for commercial security, requires that data of interest be modified only by authorized well-formed transactions (WFTs) with the concept of separation of duty determining who can perform what transactions and make what alterations [Lee88]. In doing so essential roles are identified, users are authorized to non-conflicting roles and specific kinds of data items are modified by authorized transactions acting on behalf of users acting in appropriately authorized roles [Lee88].

The seminal work in this area was done by Clark and Wilson [CW87] although there had been an earlier recognition of the different security requirements in different environments (see for example Lipner's [Lip82] paper on realizing the same using a lattice model or Chalmers [Cha86] on the differences between military and private sector security concerns). Since then several works have attempted to meet the specified commercial integrity requirements as specified in the Clark and Wilson model.

Lee [Lee88] proposes the realization of the Clark & Wilson model using mandatory categories and the concept of partially trusted subjects. Lee's approach is to extend traditional models (such as Bell & LaPadula's) to enforce the requirements of commercial security in which mandatory categories control unauthorized modifications while partially trusted subjects control unauthorized transactions. Along with a given set of rules, it is shown that the requirements of commercial security as specified in [CW87] are met. (Appendix A offers further details.)

Karger's [Kar88] work aims at realizing separation of duty using *secure capabilities* to enforce integrity. In this formulation, each data item of interest carries with it the audit information necessary . enforce separation of duty. Making an object's audit information part of its secure capability makes this information readily available This strategy facilitates performance improvement given that the same information does not have to be extracted from the audit trail.

Sandhu [San91] discusses automated separation of duties with the use of roles and transactional expressions. Imposing constraints on these expressions facilitates ensuring separation of duty. An important extension of this is to allow for substitution of attribution, something not incorporated in the original model.

Thomsen's [Tho91] work models roles using type theory and enforcement in the design and implementation of a medical delivery system. Domain definition tables are used to associate users and their roles.

The association of O-O principles and roles has been done by Ting et. al. [TDH92]. The idea is to offer segmented object access via method authorizations. The effect is the distribution of object interface among different users accessing object information.

Concerning role organization, several different organization structures have been suggested. Rabitti et al. [RWK88, RWBK91] propose a lattice organization. Sandhu [San91] suggests use of Ntrees [San88, San89] while Ting et al. [TDH92] proposed a hierarchical organization. Using a hierarchical organization of roles, it is possible to analyse role capabilities after the instantiation of the hierarchy. Baldwin [Bal90] who refers to roles as *named protection domains* (NPDs) proposes privilege graphs for role organization.

2.6 Summary

In summary, there are different security requirements between different environments which may result in differing emphasis placed on confidentiality, integrity or both. In military systems, these requirements are driven both by the need for secrecy and integrity whereas in commercial applications, they are integrity-driven. The Clark and Wilson model is an attempt to meet the commercial-specific concerns for information security. In doing so, the model prescribes a series of properties that must be satisfied in the definition and processing of data and operations in a database environment aimed at meeting these specifications.

To meet commercial integrity requirements, in the words of Lee [Lee88], requires that we identify the required roles in a system, distinguish which of those roles are conflicting, and ensure that no use executes more than one role from a conflicting class. Specific kinds of data (referred to as constrained data items, CDIs) are modified by specified transformations acting on behalf of some user acting in some authorized role. These transformations (TPs and IVPs) are designed to ensure CDI integrity. Moreover, execution of these transformations must preserve the principle of separation of duty.

Roles form an important component of the Clark & Wilson model, hence they form a substantial element for investigations that address commercial security. Roles, their definition and management are important for realizing that the model properties are met.

CHAPTER 3

AN OBJECT MODEL

3.1 Introduction

Our formulation uses Object-Oriented (O-O) principles to model entities in the processing environment. Therefore, we need an O-O framework within which to operate. This chapter presents some O-O concepts which, in our view, can be incorporated within any O-O model. This is necessitated by the fact that there exists no standard O-O model. Although there are efforts aiming at such standardization [FKMT91, LAC$^+$94, Cat94], realizing a standard (e.g. ANSI, ISO, etc.) still appears distant.

It is not our intention to specify an O-O model fully, for such a task would be another thesis by itself. We realize that to enunciate such a model to completeness requires more than its conceptual form. Therefore, such a task is beyond the scope of this thesis. Moreover, given that O-O modeling is not the focus of this work, we believe that a conceptual "model", with emphasis on the abstract concepts of concern, will suffice. Hence our task here becomes the presentation of a conceptual framework and properties important to our modeling. In true O-O modeling, this framework may not qualify as a model. However, for purposes of this thesis, use of the term "model" should be understood to refer to the framework presented here.

Our model incorporates key properties from the O-O world such as types/classes, complex objects, aggregation, inheritance, encapsulation, message sending, methods as the only means to manipulate objects, etc. In addition, we introduce the concept of *object history* as part of an object state. This keeps track of object manipulation information. Object history is necessary where audit information is required for processing. We will define object history formally and demonstrate its incorporation into the object state. With respect to executions, we distinguish between two forms:

ordinary and transactional. The former are executions as usually defined in the O-O models, while the latter incorporate transactional properties. Transactional executions are required in commercial security environments.

At this point, we must point out that we aim at introducing certain concepts that will be of use in our formulation of commercial integrity. Therefore, although we propose a manner of incorporating proposed concepts into O-O modeling, these proposals need not be seen as prescriptive. As abstract concepts, the designer of an application would adapt them to suit the application. For example, although we propose that object history (see section 3.3.2) be considered a component of the object state, there is no reason why it cannot be modeled in another form. For example, in our proposal, a history attribute, which is updated every time the object is accessed, keeps track of this history. The same concept of history can be realized by defining an object as a quadruple (identifier, type, state, history). Therefore, ours need not be the only manner of modeling and capturing notions suggested here. What is important is that, using some approach that is consistent with O-O principles, we capture the abstract meaning of the concepts proposed herein.

In the next section we present some O-O basics and the rationale for the choice of an O-O modeling approach. The principles discussed here have generally been accepted in the manner in which they are defined. While there is no standard object model, definition of concepts as used here seem to have been accepted (see for example [Dit90, Kim90, LAC+94, Cat94]) in the O-O modeling world.

In section 3.3 we formally present our O-O model with features similar to many others in the literature (see for example [LVV88, Osb89b, Dit90, Kim90, ABD+90, FKMT91, LAC+94, Cat94]) but tailored to suit our envisioned application environment: commercial database security. We present the concept of object history and its relationship to *object state*. The object state itself is determined by *object value* which is determined by the values of an object's attributes. Object history, as defined here, is intended to capture *audit* information associated with an object's manipulation. It is important where object access and processing is dependent on past history or where there is periodic system audit to ensure the correctness of executions on objects. Further, distinction is made between *bounded* and *unbounded* object histories. Bounded histories have finite length and pertain to objects with finite processing (e.g. cheques, vouchers, etc.) while unbounded histories have infinite length (e.g. audit trails, bank accounts, etc.).

Another important inclusion is the concept of transactional executions on object state. There is salient agreement that object state manipulation takes place via operation executions. However, there is no requirement that such executions be transactional. In our formulation, we introduce the possibility that such executions can have transaction properties. *Transactional executions* are necessary in our anticipated processing environment which requires well-formed transactions executing on objects. Hence the operations that manipulate objects must be transactions themselves. Alternatively, they could be invoked within some transactional operation. Chapter 4 develops this concept further.

Section 3.4 presents the summary and what we view as key contributions of this chapter.

3.2 Some O-O Basics & Rationale for O-O Choice

Object Orientea (O-O) database systems have evolved in an attempt to approximate the modeling of real world entities. They offer a major advantage over traditional relational systems as they capture more real world semantics, a fact that makes them better at modeling complex entities. They find applications in complex modeling environments such as computer aided design/manufacturing (CAD/CAM), geographic information systems (GIS), very large scale integration (VLSI), etc. Modeling power comes at a price though. It is more difficult to specify a clean, well-defined and universally acceptable model [PÖS92].

O-O databases support conventional database functionalities such a3 persistence, concurrency control, recovery, some form of storage management that includes indexing, an *ad hoc* query facility, a provision for schema definition and evolution, etc. O-O databases also incorporate concepts from O-O programming which include complex objects and aggregation, types/classes and extensibility, inheritance and class/type hierarchies, and encapsulation and polymorphism [Dit90, Kim90, ABD+90, FKMT91, PÖS92]. In general, these concepts fall into two key orthogonal categories, namely: *structure* and *behaviour* [DHP89, PÖS92, FKMT91].

In O-O systems each entity is modeled as an object with a unique identifier. An object has an identifier, behaviour and state. The identifier uniquely distinguishes an object from all other objects in a system. The behaviour and state are determined by the *type* to which the object belongs [PÖS92, Cat94, LAC+94]. The object state, in turn, depends on the values of a set of properties (e.g. attributes) [Cat94, LAC+94] which determine the state characteristics. Object state characteristics are inherently related to object structure. Behaviour, on the other hand, is determined by the operations that can be executed on the object.

The type specifies the characteristics of its *instances*. These include the behaviour and the nature of state. Behaviour, as seen above, is determined by the operations on the instances while the state depends on the values of the properties of the instances. The *class* concept is related to the type. However, as Loomis et al. [LAC+94] point out, the class connotes an implementation while a type specifies the abstract aspects pertaining to objects. A type can have many implementations and hence more than one class [LAC+94, Cat94].

A *type* defines the structure and behaviour of its objects. Objects of a given type are referred to as instances of the type. The structure is defined by type attributes and their domains. This definition allows for complex instances since the domains of attributes can be complex or simple. In general, complex attributes have complex domains while simple attributes have simple domains.[1] The *behaviour* is defined to meet its specification in the type and is determined by the *methods* specified in a class. Methods operate on the instances or their properties on being invoked by corresponding messages sent to an object.

In general, the set of messages that a type responds to defines its *interface*.[2] O-O databases allow for *extensibility* which ensures that user-defined types and classes are handled in the same manner as system-defined ones [Weg90]. Extensibility allows for modification of existing structure, behaviour and associated constraints; it permits the rearrangement of existing instances with possible re-assignment to new classes/types [FKMT91].

O-O databases facilitate incremental development by allowing for reuse of existing definitions and code through *specialization* and *aggregation* [SS77]. The former facilitates creation of new types from existing ones via the *inheritance* mechanism which uses definitions of existing types and extends and/or redefines them. The new type is said to *inherit* the said properties (attributes, operations and relationships) from the existing type. *Aggregation* allows the composition of existing type to form new ones.

[1]A simple attribute has simple values such as a number, string, or any other value specified as simple. A complex attribute can assume complex values such as another object or an aggregate of such objects or structures.

[2]Loomis et al. [LAC+94] include type relationships, operations and attributes of which programs can get and set values in their interface. While we have no problem with this approach, we are interested in the interface as determined by the operations on the objects of a given type.

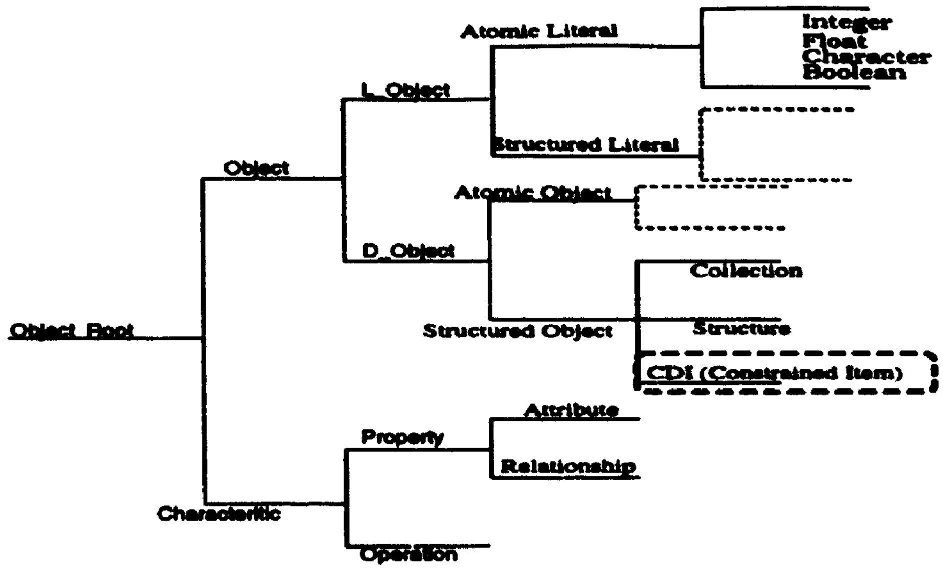

Figure 3.1: Basic Object Model Hierarchy

As new types are created via the inheritance mechanism, a hierarchy of database types results which is a directed acyclic graph. Depending on whether generalization is allowed, the hierarchy can grow in either direction of the graph. Just as we have inheritance hierarchies, so also can we have aggregation hierarchies.

All communication between objects in the database is via *messages*. Messages invoke methods which manipulate the objects as defined in the type hence providing *encapsulation*: we cannot access the representation other than through the class method interface. To ensure *Polymorphism*, instances of different types can receive the same message but respond differently depending on the type of the receiver of the message.

3.3 An Object Model

Since there exists no standard O-O database model (see for example [ABD+90, Kim90, Dit90, FKMT91, LAC+94, Cat94]), we need to specify a framework for our formulations. While the concepts incorporated here are similar to many other models, notably [AH90, LVV88, Kim90, LAC+94, Cat94, PÖS92] and others, this specification will put into clear context both the model and its properties in our conceptual framework.

We have tried to keep the model simple presenting mainly those concerns unique to our intended application. These include the nature of objects and their uniqueness (section 3.3.2); classes, types and the manner of their administration and utilization (section 3.3.3); extensibility and its bearing on database extension (section 3.3.4); methods, their nature and effect on database objects (section 3.3.5). Others include message passing, encapsulation, polymorphism and persistence.

Our application environment anticipates transactional operations; hence methods, the only change agents, have been formulated in a manner suitable for the incorporation of transactional properties. What we do not present here is a query facility for the model. It is our contention that it is beyond the scope of this work. Not that querying would not be necessary, but that it is possible to incorporate (in the context of the application) a suitable query facility should need arise. However, it suffices to note that the transaction formalisms could be extended in such a manner that every query is handled as a message.

3.3.1 Core Types & Classes

The model we propose has a lot in common with models proposed in [PÕS92, LAC+94, Cat94]. Its core types are discussed in this section and figure 3.1 (adopted from [Cat94, LAC+94]) merely demonstrates how these concepts can be incorporated in a given model hierarchy. For purposes of this work, Cattel et al.'s model [Cat94] provides the framework for our formulation. The classes include **Object_Root**, as root of the hierarchy. The **Object_Root** type defines all the basic properties and operations available for all types in the type hierarchy. It has two key subtypes, namely, *characteristic* [LAC+94] and *object* [LAC+94]. The former represents factors that determine the state and behaviour of objects. These include the operations and attributes that may be associated with objects in the database. The characteristic type thus has two subtypes, viz: property and operation.

The object subtype, on the hand, represents the kind of objects that can be found in the database. The object type has two subtypes: *D_object* and *L_object.* The D_Object has two subtypes: *atomic object* and *structured object.* The L_object subtype also has two subtypes: *atomic literal* and *structured literal.* Of major interest to us is the *structured object* type. This type has three key subtypes: *collections, structure* an. *constrained data item (CDI).*

The **atomic literal** type is a subtype of the **L_Object**. This type, along with its subtypes represent properties of pre-defined atomic types which are used as "building" blocks for the construction of new types. Subtypes of the atomic literal type include literals, integers, floats, Boolean, etc. There is also the **structured literal** under which user-defined atomic types can fall.

The **D_Object** type has two subtypes: *atomic object* and *structured object*. The former has three instantiable subtypes, namely, *type, exception* and *iterator* [LAC+94]. The **Structured Object** type provides basic properties for structured objects and instances of its subtypes. Like the atomic type, the instances of this type and its subtypes have attributes with domains from any valid type within the system. The **Collection** type is a subtype of the *structured object* type and it specifies the properties and operations necessary for instances of collections. Such collections can be *sets, bags, lists, sequences, etc.* [LAC+94, Cat94].

In extending Cattell et al.'s hierarchy, we add one type namely the *constrained data item* (CDI) type which is of greater concern to us in our subsequent formulation. The CDI type and its subtypes offer properties essential for objects ensuring object integrity as required by the Clark and Wilson model [CW87]. Termed constrained data items (CDIs) [CW87, Tho91, San91], these objects and their operations are designed such that operation execution on the objects assures object integrity. Essentially, the key differences between this type and other types is that CDIs keep track of their histories, they have transactional operations and, on execution, these operations update the the history of the object they are operating on. The history, its manipulation and the transactional properties of CDIs are essential for object processing and serve to assure object integrity. We shall revisit this issue later in this and the next chapter.

3.3.2 Complex Objects

An object is the basic unit of access in our model. It is a unit of information which is handled as an individual entity. It is the information bearing receptacle and its access facilitates access to the information contained therein. An object is specified by its unique identifier (identity), type (see section 3.3.3) and state. Object information is held in its underlying object structure and is accessible only via its interface. An object is specified by three components:

Definition 3.1 Object: *An object (o) in our O-O model is a triple:* **object identifier,** *its* **type** *and its* **state** *[LVV88, KC86, PÖS92], i.e. $o = (oid, type, state)$ where* **oid** *is the unique identifier for the object,* **type** *is the type of which the object is an* *instance and* **state** *is determined by the values of the object's properties.* **oid** *and* **type** *are drawn from countably infinite universes, $\mathcal{ID}$ and $\mathcal{T}$, respectively.* □

The identifier serves to distinguish an object from all others in a system, hence the requirement for its uniqueness. There can be many objects belonging to the same type (or even different types) with the same value. Objects that are instances of the same class have the same structure. Hence it is the identity that distinguishes object uniqueness. Identity is also used for testing for identity, equality and shallow equality [KC86, Osb89a] of objects.

Identity is very important in our envisioned application environment. For instance, no two cheques in the commercial world are the same. They may come from the same company, authorized by the same personnel and even be of equal amounts payable to same the payee. However, each one must be *unique*. Consequently, every entity must be unique in our conceptual model.

Depending on the type structure, an object can be simple (e.g. the number 2), or complex (e. g. a family object with attributes father, mother, children, etc. which are themselves complex). It can also consist of distinct objects. In general, an object can be *atomic*, simple (e.g. an integer), tuple structured (e.g. a person described by distinct attributes) or collection structured (as in the case of (say) a group object described by the set of identifiers of its members). Objects can be structured from any valid types and their subtypes in a given system.

The state of an object is determined by the values of its properties (attributes and relationships). While both attributes and relationships are important in the determination of object state, our concern here focuses on attribute values as opposed to relationships. For further reading on dealing with relationship values see [LAC$^+$94, Cat94]. Thus, in our formulation, the state of an object is determined by the value of its attributes. Let *attr* and *attrval* be the attributes and their attribute values, respectively. Object state is defined as the set of all attribute-value pairs of the object. Formally:

Definition 3.2 *Object State (O_state): The state of an object is determined by the* **value** *of its attributes, i.e. O_state={(attr,attrval)}.* □

Depending on the nature of an application, an object in a system can carry, within its state, its history. Such an object will have a *history* and *non-history* components. F r example, each CDI object in our model must maintain its history which records operations and the nature of the effects they have on the object whenever the object is accessed. The history thus captures essential *audit* information for the object.

Since audit information varies from one application to another, its form is application dependent and can be modelled in a manner the suits the application.

Since object history is part of the object state, we can partition the object state into two components: *history* and *non-history*. Let {NH_attr} and {NH_attrval} be the set of attributes and their corresponding values, respectively, of the of the non-history attributes of an object. Further, let (NH_attr, NH_attrval) be some non-history attribute-value pair while {(NH_attr, NH_attrval)} is the set of non-history attribute-value pairs of a given object. Then the non-history state of an object is defined as:

Definition 3.3 Non-History State: *The non-history state of an object is the set of non-history attributes together with their respective values, i.e. NH_state={(NH_attr,NH_attrval)}.* □

Similarly, we can define the history state of an object using the set of the history attributes {H_attr} together with their respective values {H_attrval}. Let (H_attr, H_attrval) be some history attribute-value pair and let {(H_attr, H_attrval)} be the set of history attribute-value pairs. Formally:

Definition 3.4 History State: *The history state of an object is the set of history attributes together with their respective values, i.e. H_state={(H_attr,H_attrval)}.* □

The history state component represents the object history (we define the form of this history shortly below) while the non-history component is the rest of the state other than that corresponding to the history.

Hence object state can be defined as:

Definition 3.5 Object State: *The state of an object O_state= H_state ∪ NH_state where H_state and NH_state represent history and non-history states, respectively.* □

The value of the H_state of an object carries the object's *history*. The number of history attributes required for any given situation, and hence the nature of history information carried in each attribute value, depends on an application and the application designer. At the very minimum, we require at least one attribute to keep track of the H_state. In this formulation, the history (other situations may choose different abstract definitions of such history) itself is an *ordered sequence* of events where an event itself is defined as:

Definition 3.6 Event: *An event is a quadruple $e = \langle evname, act, uid, time\rangle$ where* **evname** *is the event name,* **act** *is the nature of the action,* **uid** *is the identity of the* *subject executing the event, and* **time** *is a chronological indicator of the time of the event. Given two events* e_i, e_j *we say* e_i *precedes* ($\preceq$) e_j *if and only if* $e_i.time \leq e_j.time$.

□

Each event stores necessary audit information resulting from the occurrence of the event. This is stored in the *act* component of the event. The exact nature of this information is application dependent. Given all object histories, one can construct the system audit record by ordering the events according to their time (**e.time**) of occurrence and appending the object name or oid.

Consequently, object history is defined as:

Definition 3.7 History: *A history* $H = \langle e_1, e_2, e_3, \cdots \rangle$, *where each of the* e_i*s is an event. A bounded history is of the form* $H = \langle e_1, e_2, e_3, \cdots, e_n \rangle$ *where* n *is finite. It is unbounded otherwise.* n *and* H *are related via the size function, i.e.* $n = size(H)$.

□

The history state of an object is dependent on both the non-history and history object states given that it records information pertaining to both the non-history and history object states. The nature of history information is application dependent, though.

From the foregoing definition of object history, it follows that the value of a history attribute will be a sequence of events, i.e. every H_attrval will be of the form $\langle e_i, e_j, e_k, \cdots \rangle$.

Object history can be bounded or unbounded. For example, a cheque has bounded history while an audit trail's history is unbounded. This nature of object history partitions our object base into two distinct categories: those with bounded and those with unbounded histories[3]. Objects with unbounded history must specify what "length" (e.g. time interval, number of entries, etc.) of the history must be retained or made visible/available to applications. For instance, the audit trail record in a firm may be maintained over a period of one year or with so many entries. There is likely to be a problem should the size of the history be left to grow indefinitely without continuous or periodic *sanitation*.

All objects in our models are *persistent* and live beyond the applications that create them. Object *destruction* must be carried out *explicitly*. Once an object is destroyed, it cannot be accessed in the database any more and any references to it must be updated. However, like any database operations, this destruction action

[3]As we shall see in the next chapter, these are referred to as documents and accounts, respectively.

action must be recorded in the system audit trail. This requirement clearly conforms to that in the real world where documents are destroyed once they outlive their useful lives. However, some objects are *immutable*. They cannot be destroyed.

3.3.3 Types & Classes

A type specifies the form and behaviour of its instances. It is specified by its name, structure and method list. Formally:

Definition 3.8 *Type: A type* **T** *is a triple:* **(n, s, m)** *where* **n** *is the name of the type,* **s**, *is its structure and* **m** *is the method list applicable to the type.* $n \in \mathcal{N}$ *is unique and drawn from a universe of infinitely many symbols* $\mathcal{N}$. □

The **name** identifies the type. It must be unique in a given system. While there may be other means of uniquely identifying the type, such as some *type identifier*, we do not address that matter here. We point out that a type must be uniquely identifiable in a system. In our case, the name serves this function.

The **structure** determines the structuring of the instances of the type and their values. It is composed of a list of instance variables and their domains. Domains can be simple or complex, hence instances can be complex or simple depending on the domains of the attributes. A type structure can take any of the three forms: simple, tuple, set or sequence.

A type structure is defined by a list of name-domain pairs called *instance variables*. The instance variable name must be unique within the structure while the domain can be any legal type in the system. Instance variable name uniqueness ensures that the instance variable can be accessed within the type without ambiguity.

Given some type with instance variables $(a_1, \cdots, a_n)$ and some instance o of the type, we use the "dot" notation (•) to refer to the individual instance variables. Hence $o \bullet a_i$ will refer to the value of instance variable a_i within object o. Where object are complex, use of the dot notation leads to path expressions.

Methods (see section 3.3.5) provide the only means to manipulate the state of instances of a type. Methods can either update (alter) or read the state of an instance via the attribute values. We denote the universal set of methods in the system by $\mathcal{M}$.

Note that different types can have the same structure but different method lists just as different types can have the same method list but different type structures.

Moreover, since type definitions are treated as objects, it is possible to do identity testing between two different types using object identity.

Example 3.1 **A type definition according to definition 3.8, using the syntax of [Osb89a], is of the form:**

```
Name:
                CHEQUE;                 /* The name of the Class n */
Structure: {                            /* Name-domain pair list of instance variables */
                PAYEE: String,
                PAYEE_ID: String,
                AMOUNT: Currency,
                SIGN_1: String,
                SIGN_2: String
                };
Methods: {                              /* Method list for manipulation of instances */
                clerk,supervisor
                }
```

□

In our model, the type is a template for objects and defines the characteristics of the objects of that type. A type definition can have either an empty structure or an empty method list. For instance objects that carry out computations, can have just the method list while objects used merely for modeling that need no access, may have an empty structure.

A **class**, according to [Cat94, LAC+94], connotes an implementation. Therefore, we can have different classes realizing a given type definition. Suppose we have some class implementing the type definition of example 3.1. Such a class specification will exist separate from its *extension* (instances of the type). Defined this way, a class is a template of its instances. The class extension, on the other hand, is the collection of instances of the class and exists only via the notion of *instantiation* of a class. An undefined class cannot be instantiated; hence the existence of some instance presumes the existence of the class of which it is an instance. Consequently, no class extension exists without the corresponding class definition.

Object creation takes place via its class and class extension; the class specifies the object structure and behaviour while the extension facilitates instantiation. A class extension while being a collection of objects differs from any arbitrary collection in that it collects all instances of the same class. An arbitrary collection, on the other hand, can collect instances of arbitrary classes. It follows that whereas an object can be associated with different collections, it must be *uniquely* associated with some class extension.

3.3.4 Extensibility & Types

Extensibility allows for the addition of new types, redefinition of existing types and re-assignment of instances to new types with which they were not associated before [FKMT91]. Redefinition of existing types and addition of new types are elements of schema evolution, while instance type re-assignment can be seen as instance evolution [FKMT91]. With extensibility, newly defined types are treated as any system defined ones.

New types are defined via specialization [SS77]. In specialization a new type *inherits* properties (attributes and methods) of an existing type, adds more properties and may redefine the inherited ones. The new type is a *subtype* of the old type and with its properties being a superset of those of the old type.

Definition 3.9 *Type-Subtype: A type X is a specialization (subtype) of another type Y if the property set (attributes and methods) of Y is a subset of the property set of X with the two being related via the inheritance mechanism.* □

Note that the implementations of properties in the type and subtype need not be the same. It is not sufficient to have a subset property relationship between type for specialization since types can have a shared interface. The two must be related via the inheritance mechanism. Inheritance allows for redefinition of inherited properties.

Inheritance is a means of sharing type definitions and implementations via defined/implemented structure and behaviour. It facilitates not only sharing of these properties but allows their extension and/or redefinition. This allows for reuse and supports incremental development. In the words of Krakowiak et al. [KMV+90] "inheritance supports conceptual economy" via reuse. While shared behaviour and their implementations are distinct issues, we do not make the distinction in our model. This distinction, in our view, is more of a syntactic issue as opposed to semantic.

Our model allows for *multiple inheritance* in which an new type inherits properties from more than one type. There can be name (both attribute and method name) conflicts with multiple inheritance. To resolve such conflicts we suggest utilizing the implementation and structures of the names in conflict. Where there is conflict between two properties with the same implementation, we suggest name selection based the order of types in the inheritance list. Where there are different implementations of properties, but with the same output result, the same strategy first-in-list conflict resolution strategy is employed. However, where the different implementations produce different output results (as in the case of methods with similar names but with different computations) we defer resolution by the user. A user can then choose one or both of the properties in conflict. Renaming is necessary where retention of both is required.

Inheritance results in an *inheritance (specialization) hierarchy* with the *is-a* relationship between types and subtypes. The hierarchy is a directed acyclic graph rooted at some base type (see figure 3.1). Since we do not allow *generalization*, our type hierarchy "grows" in one direction. Inheritance semantics emphasize *self reference* in response to messages (see section 3.3.5), i.e. data and operation look-up starts at the receiver.

O-O databases facilitate incremental development by allowing for reuse of existing definitions and code through *specialization* and *aggregation* [SS77]. Specialization facilitates creation of new types from existing ones via the *inheritance* mechanism which uses definitions of existing types and extends and/or redefines them. The new type is said to inherit the said properties (attributes and methods) from the existing type.

Defining a type via the inheritance mechanism involves specifying its supertype.

Example 3.2 Example of a type definition using inheritance:

```
Name:
            PERSONAL_CHEQUE;           /* The name of the Class n */
            Supertype: CHEQUE;         /* Name of supertype from which
                                       to inherit properties */
Structure: {                           /* Name-domain pair list of instance
                                       variables */
            TAX_PID: String,           /* Extra identifier for taxation */
            DEPT_ACCTNUMBER: Integer   /* Account from which to pay */
            };
Methods: {                             /* Method list for manipulation
                                       of instances */
            cashier
            }
```

□

The **PERSONAL_CHEQUE** type, aside from the properties in its type definition, will also have those from the type **CHEQUE**. On compiling type definitions, these properties will be augmented.

As new types are created via the inheritance mechanism, a hierarchy of database types results which is a directed acyclic graph. Just as we have inheritance hierarchies, so also can we have aggregation hierarchies.

3.3.5 Encapsulation, Messages & Methods

Our model implements the concept of encapsulation that allows information hiding thus separating the *public object interface* from the implementation [FKMT91, BKK88]. This facilitates the use of objects without knowing how they are implemented as an interface can have different implementations. In this formulation, interface connotes the means via which object object information is accessed or manipulated. Every type definition includes an interface specification. A class that implements some type must ensure that its interface meets the type interface specification. Given a type specification, a user need only know what message to send to cause the desired effect on an object.

Messages are the only means of communication between database objects. The set of messages understood by an object constitutes its public interface which can be seen as the object protocol [FKMT91]. This interface determines the behaviour of the associated object. This work assumes that there exists a suitable mechanism (such as signatures) which associates the implementation of an element of the interface to the specification of the element in the type definition. Hence we do not worry what constitutes this implementation as long as it meets the interface specification. Delving into implementation details is beyond the scope of this work.

Methods implement abstract specifications defined in the associated type. These operation specifications specify the abstract form to be realized by any method implementation. Each operation specification designates the operation signature, the name of the operation, the name and type of input arguments, name and type of output values as well as any exceptional conditions [LAC+94]. In the following sections we use the terms operation and method alternately.

Definition 3.10 <u>Object Interface:</u> *Given an object o and the set of database messages (regarded as the message universe in the system) $\mathcal{MS}$, the interface $OI(o) \subseteq \mathcal{MS}$ is the set of messages understood by o.* □

A subset of an object's interface is called an *interface partition* or *window*. Any subset of the messages an object responds to can be seen as a *window* into object information.

Definition 3.11 *<u>Interface Partition:</u> An interface partition $part(OI(o))$ of some object o is a subset (any subset or nil) of the object interface OI, i.e. $part(OI(o)) \subseteq OI(o)$,. Given some interface $OI(o)$, its partition $part(OI(o)) \in 2^{OI(o)}$.* □

Messages sent to objects invoke methods. A message has a name (message identifier), a sender, a receiver and a parameter list. The message name is drawn from a unique list of message identifiers, both sender and receiver are database objects, the input parameter list forms the input list for the computation while the return parameter list determines the result of the computation.

Example 3.3 **A message is of the form: (sender, receiver, name(arguments)). For example if o_1 sends a message to o_2 it may appear as: $(o_1, o_2, m_i(x, y, \cdots))$. Here he method selector m_i takes arguments $x, y, \cdots$ and is a communication between objects o_1 and o_2.**

□

Messages are polymorphic, i.e. objects react to the same message individually giving the associated respective response. On receipt of a message, a method with the corresponding name is invoked. In our model the message name corresponds to the method name. In the rest of this work, method invocation must be understood in the context of message receipt and method invocation. As well, we do not outline how this message sending paradigm is implemented.

Methods manipulate the state of an object; they update (change) or merely return information about the object state. In general, methods read object state, alter the state, invoke other methods and/or create other objects. In our model, method invocation is similar to that of [JK90]. It has the following modes of invocations:

1. a method can access object attribute(s) directly with resultant effects that it returns the value(s) and/or changes such value(s) of the attribute(s). We term such an invocation *simple* for its action does not involve invocation of other methods. Method invocations other than simple invocations are termed *compound*, *complex* or *composite* invocations. Methods that return values of attributes are referred to as *read methods* while those that alter the value of attributes are termed *write methods.*[4]

2. a method can invoke other methods acting on the same object. In general, such an invocation can have more than one level, which is generally termed complex. Such an invocation will result in a sequence that, in its general form, is a tree[5],

[4]The term *transformation methods* has been used as well.

[5]The tree representation is essential to ensure that there is no chance of cyclic invocation.

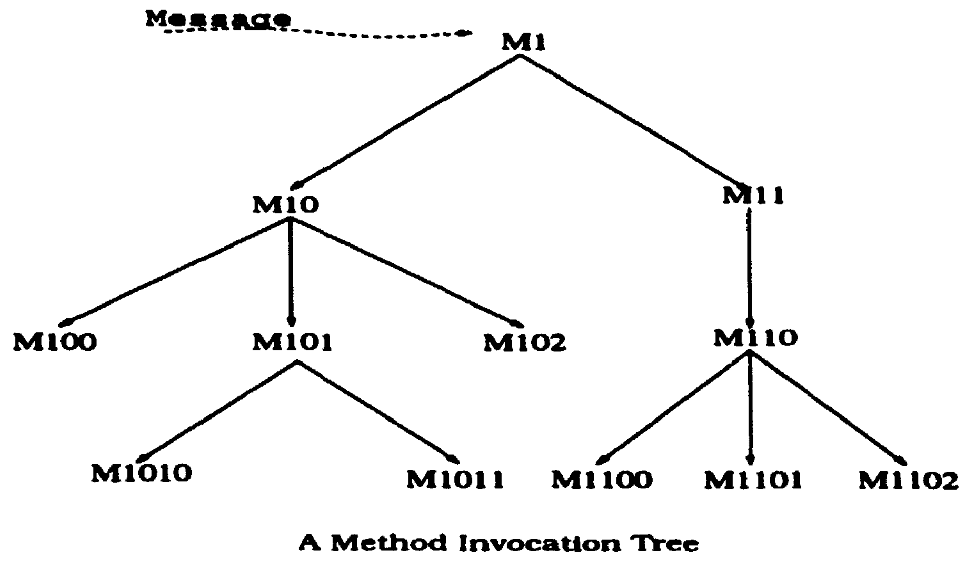

Figure 3.2: A Method Invocation Tree

termed the method invocation tree [JK90] (see figure 3.2). To distinguish such a case from the next one, we term it a *simple* method invocation tree in the sense that it involves invocation of methods *local* to one object. A method invocation tree is either simple or compound (see below).

3. a method may send messages to other objects. For instance if the computation for a particular method requires inquiry from other objects, a message will be generated. As in the previous case, such invocation can be represented as a tree spanning methods of more than one object. We term such a tree a *compound* method invocation tree.

4. a method may create new objects. This may be the case where on commit, after processing involved within some object, the method creates a new object (cf the cheque voucher system where the approval of a voucher, on receipt of all requisite signatures, creates a new object called a cheque).

Definition 3.12 Method Definition: *A method has* **name, body** *and* **message format**. *It can be represented as a triple* (n, f, E). n *is the method name which must be unique in the type,* f *is the message format and is of the form* $A_1 \times A_2 \times \cdots \times A_p \rightarrow \{R_1 \times R_2 \times \cdots \times R_q, ex_1 \times ex_2 \times \cdots \times ex_n\}$ *while* E *is the executable code of the* *method. $A_1, A_2, \cdots, A_p$ is the input argument list with p being the number of arguments. $R_1, R_2, \cdots, R_q$ is the output argument list where q is the number of outputs of the method and any of the R_is could be: a method, object, message, or nil or their combination (see above). $ex_1 \times ex_2 \times \cdots \times ex_n$ is the set if exceptions for the particular method [LAC+94]. The ex_i could be methods, messages, etc. tagged to some exceptional condition.*

Let $\mathcal{M}_c$ be the set of methods defined in a class c and $\mathcal{M}$ be the set of all methods in the database. Then $\mathcal{M} = \bigcup_c \mathcal{M}_c$. □

The arguments specify their form and corresponding classes. The message format f can be seen as the method signature [KMV+90]. It specifies the nature of the arguments (objects, messages, methods, etc.) and their form (e.g. class, sender/receiver/message name, etc.). It also distinguishes input and output arguments in the list.

Definition 3.13 *Method Invocation Tree: A method invocation tree, T_m, for a method m is a directed acyclic graph with methods as nodes. An edge $\langle m_i, m_j \rangle$ in T_m means that m_i invokes m_j. m, the method associated with the tree, forms the root of the tree.* □

Definition 3.14 *Tree Path: A path between two nodes m_i and m_k is a sequence $\langle p_1, \cdots p_n \rangle$ with $m_i = p_1$ and $m_k = p_n$ and $\langle p_j, p_{j+1} \rangle$ is an edge from the node referenced by p_j to the node referenced by p_{j+1}.* □

The path is a total ordering of nodes. Given a path p then there exists another path q such that $p \subseteq q$ and q has the root as a node. This implies that the method invocation sequence can be represented as a tree.

Definition 3.15 *Let "$\prec$" denote the precede relationship between two nodes in a given path p. A node m_k is said to be a descendant of another node m_i if $\exists$ path $p = \langle m_i, \cdots, m_k \rangle$ and $m_i \prec m_k$ in the path.* □

Example 3.4 Method Definition

Name:

CLERK;

Format:

UID × ACL × Object $\mapsto$ Object × Audit Record

Ei

Pointer to Executable Code

□

An important extension of this method definition is the incorporation of transactional properties into their execution. These are the ACID properties that ensure atomicity, consistency, isolation and durability of database executions. While we do not specify the extensions here, it suffices to note that, in our envisioned application, methods will execute as, or as part of, some transactional operation. This is due to the requirement that each operation on the state of any object in the database must be transactional in nature.

Definition 3.16 Transactional Operation: *A transactional operation, in our model, takes the object state from one consistent state to another consistent state and in the process, permanently transforms the object state. It permanently transforms the history component. Possibly, it also permanently transforms the non-history component of the object state.* □

In general, a transactional operation is of the form:

```
        pre-conditions
    IF preconditions then
        BEGIN
            { Execute operations ensuring isolation }
            IF commit conditions then
                COMMIT { operations }
            ELSE ABORT { operations }
        END.
```

Let TE and NTE be the set of operations that execute as transaction and non-transactions, respectively. We have that:

Definition 3.17 Transactional Executions: *Given a method m with an associated method invocation tree T_m, let m_{tset} be the associated methods forming the nodes of T_m. We say $m_i \in m_{tset}$ executes transactionally if:*
(1) m_i executes as a transaction, i.e. $m_i \in TE$
(2) or $\exists m_j \in m_{tset}$ such that m_i is a descendant of m_j and m_j executes transactionally. In other words, $m_i \in NTE$ and $m_j \in TE$ and $m_j \prec m_i$ in some path p of the method invocation tree T_m. □

A type in which all executions are transactional is a subset of the CDI type (see figure 3.1) when all instances keep track of their histories as well. We have the following constraint:

Constraint 3.1 *Method Invocation Constraint: Every method invocation in a CDI type or its subtype must execute transactionally.* □

Transaction *atomicity* ensures that all or none of an operation's activities have a lasting effect. This is taken care of by the transformation condition of the non-history component of the object state. However, given the requirement to update object history every time an object is accessed, there must be a permanent transformation of the history component. History component transformation must be consistent with the effect on the non-history component. Where there there is **commit**, the non-history component is transformed and commit information entered in the history component. In case of **abort**, the non-history component is not transformed while the history component records the abort information.

Example 3.5 **Consider the type definition in example 3.1. Assume we have one attribute HIST that keeps track of the history of the object, i.e.** $H\_state = (HIST, val(HIST))$. **The new type structure will be of the form:**

```
Name:
            CHEQUE;
Structure: {
            PAYEE: String,
            PAYEE_ID: String,
            AMOUNT: Currency,
            SIGN_1: String,
            SIGN_2: String
            HIST: SequenceofEvents
            };
Methods: {
            clerk,supervisor
            }
```

A transactional operation is bounded by BEGIN and END key words, i.e. anything between these two keywords must be done in the context of a transaction. It will be of the form:

```
BEGIN
  on invocation of method        /* invoke some method, any method */
  check:=false;                  /* set the Boolean guard */
  if check(UID,H_state) then     /* Check for preconditions here */
      execute method;            /* Execute some code associated with NH_state */
```

```
        commit                  /* Commit makes NH_state changes permanent */
      else abort;               /* Abort undoes changes on NH_state */
        update(H_state)         /* Update H_state accordingly */
   END
```

□

Other constraints are that such operations must be dependent on object history and must update this history on completion of execution, regardless whether the associated transaction commits or aborts. As well, when specified, the operation must update system history (audit trail). Simply said: ***an operation on a database object must update the object state.*** This because object history is part of the object state. Hence, more precisely, an operation on a database object must update, at the very least, the *H_state* of the object's *O_state*. We shall revisit this matter in chapter 4.

3.4 Summary & Key Contributions

In this chapter, we have outlined an object-based model that incorporates key O-O concepts. As a conceptual framework our formulation incorporates known O-O concepts such as complex objects, unique identity, types, type (inheritance) hierarchies, inheritance, subtypes, message passing, method invocations, encapsulation and, polymorphism. In addition to these we introduced ***object histories*** and ***transactional executions*** of methods.

An object interface in traditional O-O models is determined by the messages it responds to. A message, in turn, activates the respective method associated with the object. An important variation introduced in our model is the possibility of method executions being transactional. Thus we introduced a type (CDI) in which all executions have transaction properties and all instances keep track of their histories. Method of the CDI type and its subtypes are confined to execute within some transactional framework.

Issues of schema definition and management were not addressed. Although they affect our intended formulation, it suffices to note that a schema is handled the same way as an object. Hence it is subject to similar regulations as other objects. Schema information manipulation is similar to object information manipulation given that schema integrity is as important as object information integrity. Others include data manipulation and data definition languages which we consider to be beyond the scope of this work.

The two key contributions presented in this chapter include the concepts of object history and transactional executions, both of which are new to O-O modeling. The former helps to keep track of audit information pertaining to objects. As well, history, as we shall show later in this thesis, is important in the task of access control where separation of duty is necessary. Transactional executions are also important in ensuring that executions have the necessary properties that assure object and operation integrity.

CHAPTER 4

TRANSACTION MODEL

4.1 Introduction

Commercial security [CW87] requires information bearing entities (objects) to be constrained data items (CDIs)[1] which should be manipulated by transformation procedures (TPs). TPs themselves and their effects are verified by integrity verification procedures (IVPs). Both TPs and IVPs are required to be well-formed transactions (WFTs). The requirement of transactional nature of executions forms the basis for formulation of a transaction model in this chapter. We use the Object-Oriented (O-O) paradigm and the O-O model of chapter 3 to define both the CDIs and the WFTs for manipulation of the CDIs. Due to the complexity of the envisaged processing environment, the traditional transaction model will not be adequate, in our view. It will be limited in its ability to handle the nature of operations in our applications. This is in the same light as in applications which require models which handle complex operations, support long running activities, facilitate concurrent access to complex objects, permit transaction nesting, etc. This chapter will discuss the said shortcomings and propose a model which incorporates *desirable* features suitable for our application.

The envisaged environment, while exploiting O-O principles, models executions as transactions. Within each CDI type/subtype, we define transaction and transactional executions (see page 46 in chapter 3) which may or may not be *ordered*. In theory, these could be invoked independently of each other. However, one or more orders of execution of these transactions could also be prescribed. Each of such specified orders of execution is referred to as a *script* [WR92]. A script specifies the nature of processing associated with an object of the associated type. In any given type, one can

[1]At least those that require preservation of integrity.

have more than one script. Moreover, since all script executions will be transactions themselves, along with each script, we must define a related *compensating script*. The latter are defined such that they can undo the effects of the associated script. This choice is due to the transaction *nesting* feature in our model in which child transactions can commit independently of either the parent or other child transactions. Compensating scripts undo the effects of such committed child transactions should the parent abort.

This chapter is not a treatise on transactions execution *per se*, but rather an emphasis on the manner transactions can be ordered to achieve a desired outcome. Hence by defining transactions and transactional executions in a specified manner based on desired outcome, we can ascertain the outcome *a priori*. Such execution outcomes can be designed to ensure the database is left in a valid state. This then provides a means of assuring that the transactions will leave the database in a valid state. It can also be tailored to specific integrity requirements. Therefore, the focus of this chapter can be seen as prescribing the manner of specifying execution orders to achieve desired outcomes.

In section 4.2 we discuss further the nature of our operation environment and the security requirements. In particular, we discuss concepts of IVPs, TPs, WFTs and CDIs with the intention of crystallizing the transactional requirements of our executions. We relate these to O-O modeling principles and distinguish between ordinary objects in the database and those of the type CDI. Of the latter, we have two types which we call *documents* and *accounts*. Documents have a finite history while accounts can have unbounded histories.

Both of these are processed based on some script definition. Constraints specified on the script take into account current history as well as other correctness constraints that may be required. Since we utilize O-O concepts in defining our database objects, we dwell for some time on the manner in which the *CDI* type is handled in our system.

Section 4.3 offers transaction theory overview. Here we discuss the traditional transaction model, its shortcomings and summarize some approaches intended to overcome these deficiencies. The key issues addressed, among other things, are the concepts of nesting, handling complex operations and data, etc. Some approaches to addressing some of these shortcomings will be covered in this section.

We present a transaction model in section 4.4. Given the complex nature of operations and data in our envisaged operation environment, the model addresses the shortcomings of the traditional model. It is based on the nested transaction model

[AA92] and uses the concept of scripts [WR92] to specify the order of execution of the associated transactional executions. A script specifies the order in which the TPs that manipulate CDIs are executed according to processing requirements. A CDI can have more that one script, each one designed for a specific processing goal pertaining to the CDI. Further, since the operations are transactional in nature, undoing their effects in a database requires compensating actions. These are specified as compensating scripts which are part of the definition of an object of a CDI type or its subtype. Generally, a script defines the flow of control in processing objects of such a type. Having proposed a transaction model, we demonstrate its ACID properties in section 4.5.

Section 4.6 offers a running example that we shall be using for the rest this work. It is an illustration of a framework in which our modeling could be applied. It is based on an inventory management situation in which information about the processing is stored in constrained data items. We shall formally describe the environment, the procedures involved as well as the object necessary for realizing the said processing. Finally, we present a summary and the key contributions arising from this chapter's discussion in section 4.7.

4.2 Operating Environment

As mentioned in the introduction to this chapter, our operating environment assumes requirements for commercial security. Among these are the specification of data items (at least those whose integrity must be ensured) as constrained data items (CDIs). These must always be in a correct state. Once initialized, their manipulation must ensure that it takes them from one correct state to another correct state. The operations that access CDIs are called transformation procedures (TPs) and integrity verification procedures (IVPs) which must be transactional in nature. Should the input be in some incorrect state, they must ensure that the output CDI is in a correct state; otherwise the operation is rejected. Indeed, the requirement in the Clark and Wilson Model [CW87] is that if a TP takes in an unconstrained data item, UDI, its output must be a CDI, or else the process is aborted. Both TPs and IVPs must be well formed transactions in their execution.

The behaviour of IVPs, as specified by the Clark and Wilson model can be subjected to different interpretations. For instance, it is possible to regard IVPs as procedures that ensure th t the interaction between TPs and associated CDIs will guarantee the correctness of the state of the CDIs. Yet another view is that IVPs play the role of an audit program [AAL+93] that ensures that TPs maintain the specified validity of the CDIs they manipulate. While both these interpretations may be correct, the former could be overly costly given that IVPs will play a role of verification which is itself a major area of study. In our environment, IVPs play the role of audit programs that assure the correctness of the CDIs and hence serve to draw the attention of system managers to any anomalies.

We assume the object model of chapter 3 in which the CDI type and its subtypes impose integrity requirements as required for commercial integrity. We distinguish two types of objects in our environment: *documents* and *accounts*. Both types of objects are constrained data items (CDIs) as explained on pages 22 and 34. The former include types such as vouchers, cheques, bank statements, orders, notes, etc. These have a short processing life accomplished by a finite number of steps. Hence their histories are finite. Subject to processing requirements, such as audit trail and object history updates, this category of objects can be archived after the conclusion of their processing. The duration of such archival periods will be a matter of system policy.

Definition 4.1 *Document Objects: A document object is an instance of the CDI type or one of its subtypes with a finite history based on associated script(s).* □

Account type objects include bank accounts, audit logs, etc. These can be seen as holding the "journal" of activities pertaining to document and account objects. They, in some manner, capture histories associated with document objects in the system. For instance, the audit trail could be a perfect example of such an account object since it can be seen as a manifest of the processing in a system. It could be designed in a manner that reflects the activities performed due to (say) cheques cashed, accounts debited and credited, etc. Account objects have infinite history (such as the life of a system) a finite length of which is retained by the system based on administration policies and requirements. In account type objects, we are mainly concerned with the current state which we either read or update, since the history of such objects can be unbounded. The state of the object, however, depends on what size of this history is retained.

Definition 4.2 *Account Objects: An account object is an instance of the CDI type or one of its subtypes with an infinite history. It is read and updated according to associated scripts.* □

Actions on the state of instances of a particular CDI type are specified in an associated *script* (see section 4.4). The actual processing of a particular object follows a particular execution specified by the script along with the *constraints* on the script. These constraints are intended to ensure correct script execution. Distinction must be made between the manner of manipulation of ordinary and CDI objects in the database.

The nature of manipulation of objects in our system depends on what type it belongs to. We do not prescribe the form of execution for objects other than those in the CDI type, except to note that such executions must be in accordance with O-O principles. Instances of CDI types and their subtypes, however, must obey a prescribed order of execution. This is intended to assure their integrity. These objects must be manipulated according to the scripts defined in their types. The scripts themselves are defined in terms of transactions and transactional executions.

The effect of prescribing the order of execution a *priori* via script definition is to constrain the behaviour of the CDI objects. Further restriction can be imposed to ensure that all executions that affect the state of the associated objects must occur within the context of an associated script. While there is no reason not to invoke operations on the objects outside the script specification, it is, however, necessary that update operations on CDIs be executed according to a given script in order to assure a desirable outcome. This imposition is intended to guarantee the behaviour of objects in the CDI types and their subtypes. CDIs in our formulation, can be seen as objects with *constrained behaviour*, constraints imposed by an associated script.

Consequently, definition 3.8 is modified to incorporate this constrained behaviour property. Hence:

Definition 4.3 *A CDI type is an ordinary type with the addition of a script that constrains the behaviour of instances of the type and whose structure has the ability to track history. Formally: dtype* $= (n, s, m, scr)$ *where n is that name of the type, s is its structure, m is the method list and scr is the script for instances of this type.*

□

Our processing environment assumes the object model of chapter 3. Hence concepts such as inheritance of properties, message polymorphism, aggregation in object composition and other related O-O concepts, hold. For all document objects, a special Boolean attribute indicates whether or not the document has been completely processed. A document is said to be processed once the effects of its script have become permanent, i.e. on commit or abort. In other words, a document object is considered processed when its script has executed either successfully or unsuccessfully. Scripts cannot be invoked on processed documents. However, retrieval of parts of their histories can still take place. No updates are allowed on completely processed objects such as a paid or cancelled bank cheque, a voucher whose cheque has been paid, or a delivery note whose goods have been logged in an inventory. A system designer would designate what determines whether or not object processing is complete. Undoing the effects of a completely processed object must be done by some compensating process. For example, once a cheque is paid, its effects can only be undone by another compensating credit.

The processing undergoes an *initialization* step in which preliminary settings are made before actual document manipulation starts. This step involves the creation of a copy (instantiation) of the specific document type to be processed. Such a copy would have a unique identifier (such as a cheque/voucher number) that would distinguish it from all other documents in the system. Suppose that we require the document type **voucher**. We create a copy with the structure, behaviour and script for a voucher. Execution is then triggered and it obeys the script and constraints therein. We discuss scripts and their constraints in the next section.

4.3 Transaction Theory Overview

This section gives an overview of traditional transaction models, outlines their shortcomings with respect to our modeling requirements and proposes a transaction model with desirable features. Traditional models are suitable for simple operations such as reads and writes on simple data. They face a major handicap for long-lived transactions with or without complex operations on simple or complex data. Our proposed model is script based [WR92], pertains to complex operations on complex data, allows transactional nesting [AA92], permits independent commits of child transactions and provides compensating scripts to facilitate the undoing of committed portions of a transaction.

4.3.1 Transactions Basics

Transactions are characterized by *atomicity, consistency, isolation* and *durability,* commonly known as the ACID properties [GR93]. Atomicity requires that a transaction's operations be treated as a single unit such that all of a transaction's operations take effect or none at all. Consistency requires that a transaction takes the database from one consistent state to another consistent one. A transaction's effects on the database is the net effect of the executions of its committed operations. Such an execution is correct if its operations execute correctly with respect to their semantics and the execution takes the database from one consistent (valid) state to another. For instance, in banking where transactions credit and debit accounts, a valid database state may be defined as that where the account balances are at least equal to a mandatory minimum deposit. The correctness criterion for such a transaction is the correct execution of the operations that manipulate the account.

In real life, devoting a resource to one transaction at a time is not very useful, hence transactions have their operations *interleaved* to facilitate *concurrency*. As with the single transaction case, concurrent transactions must leave the database in a consistent state. The correctness criterion is derived from that of a single transaction, i.e. it must appear as if the individual transactions executed serially, one after the other without interleaving. Termed *serializability*, this correctness criterion requires that for any concurrent execution to be considered correct, it must be shown to be equivalent to some serial execution of the interleaved transactions.

Isolation, on the other hand, is the requirement that a transaction, any transaction, sees only a consistent database state(s). Since one transaction's intermediate results may put the database into some inconsistent state temporarily, this requirement ensures that such intermediate results are not visible to other transactions. The use of *locks* facilitates this.

Finally, durability is the requirement that the effects of a committed transaction have a permanent effect on the database even in case of system failure. Concurrency control protocols ensure correctness of concurrent execution of transactions thus taking care of atomicity and correctness. Recovery protocols, on the other hand, ensure isolation and durability of the transactions.

This forms the basis of the traditional transaction model. It assumes that the operations on data will be simple reads and writes and that the data items themselves are simple, such as in entries in a relational table.

4.3.2 Shortcomings of Traditional Models

Traditional transaction models handle ordered simple read and write operations on simple data items. Examples include reading and updating simple data values such as bank balances, deposits, etc. Given the assumption of simple data items, locking and time stamping (for concurrency control) of the data items can be used without much adverse impact on processing. As well, the assumed simple executions result in short-lived transactions that are little impacted by the use of locks and/or time stamps. With short-lived transactions, we can lock data items knowing well that the duration for which locks are held will be short. As well, without short-lived transactions, time stamp management would be prone to excessive rollbacks. In general, the traditional model handles small transaction sizes with little attendant risk of deadlock, given that deadlock frequency increases with the fourth power of the transaction size [ELMB92]. Hence simple transactions like banking, airline bookings, etc. are common users of the traditional model. For concurrency, *serializability* is used as the correctness criterion with little consideration of the semantics of the operations. Moreover, the assumption of serial execution of transaction operations (acting on simple data items) leaves no room for parallelizing the operations when such a data item becomes complex.

There are problems of using the traditional model in environments with complex data and long-lived or nested transactions. Where transactions access complex data (as in CAD/CAM/OIS[2]), are long-lived or the database is distributed, the traditional model is clearly unsuitable [AA92]. We shall refer to models, other than the traditional model, as *extended models*.

For complex data items where transactions access only parts of an object, locking has inevitably a negative impact on concurrency. It may be preferable to lock only those parts of the complex data items where there are operation conflicts (see for example [RGN90]) while leaving other parts to be accessed by other operations. The resultant impact is increased concurrency where non-conflicting operations on the same object are parallelized. Locks and time stamps in environments with long-lived activities [Day93, WR92] are clearly unsuitable. The overall impact in such cases would be decreased concurrency and inevitable rollbacks and redos. Serializability, in its traditional form may prove to be too strict a criterion for correctness; hence the need for a definition of complex data serializability [RGN90] or semantics-based serializability in which operation semantics are taken into account. For instance, even when two write operations conflict, their semantics may help increase the level of concurrency (e.g. when one has two consecutive increments to a value).

[2]CAD: Computer Aided Design, CAM: Computer Aided Manufacturing, OIS: Office Information Systems.

4.3.3 Extended Models

Transaction models such as the nested transaction model [AA92], the multilevel model [WS92], the distributed model [ELMB92], the cooperative model [NRZ92], activity/transaction model [Day93], etc., are attempts at overcoming the shortcomings of the traditional transaction model. These models allow long-lived or nested transactions, facilitate cooperation among various activities within a transaction and introduce parallelism or distribution into the processing. Failure handling is also improved by limiting such failures to small portions of the transaction. This has the overall effect of reducing the amount of rollback during the processing. Yet another issue of interest, that has been addressed, is that of local autonomy in distributed environments. Local autonomy offers the advantage of maximizing local processing and reducing the overall communication costs between different sites.

The ACID properties of the traditional transaction model form the basis for many of the extended models. They are nonetheless refined to incorporate transaction semantics, take into account their long-lived or nested nature, as well as allowing for operations on complex data items. For instance, in the nested transaction model [AA92], transaction isolation is defined within the top transaction which allows children transactions to access their parent transaction's data. Hence for concurrent transactions, isolation is enforced among the top transactions. Commit protocols also vary from model to model. In some, the children transactions can commit independently of their parents in which case compensating transactions are defined to undo their effects should the parent abort.

4.4 Our Transaction Model

The basic features of our transaction model include *nesting* [AA92], *scripts* and *compensating scripts* [WR92], among others. Essentially, each transaction in our system can be nested to arbitrary levels (see figure 4.1). Hence each transaction can be represented either by a *tree* or a *node* of a transaction execution tree. A further requirement is that transaction executions on objects must execute within the order specified by the associated script. This script can be seen as a prescribed order of execution, with no transaction executing outside the order of this specified order.

Definition 4.4 *Script: A script is a specification of the order of execution of transactions pertaining to some object in the database.* □

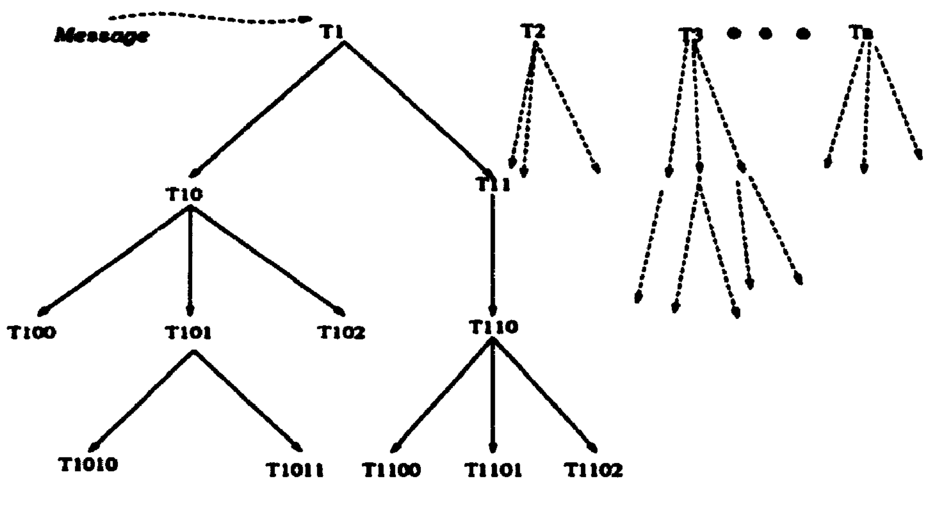

Figure 4.1: A Transaction Execution Tree

Our scheme takes into account the system processing environment and shares many concepts with other nested models [AA92, WR92]. In particular, processing in our model is guided by scripts which specify the order of execution of transactions (nested or otherwise). Script specification allows for sequential, concurrent, choice and iteration executions and can be defined to facilitate an event trigger situation for invocation of other scripts based on some dependency relationship.

In other words, a script defines the control flow of execution in a specific kind of processing. The prescribed order of execution can be subject to constraints which may be associated with the script. The constraints regulate this control flow and serve to ensure correct execution of context dependent situations. By attaching scripts to object definition, we effectively constrain its behaviour. Like Wacher and Reuter's scripts [WR92], scripts in our model determine the course of execution and the strategies of what may be long-lived *activities*. However, unlike in Wacher and Reuter's model, scripts in our model can be constrained by script specific constraints that may take into account context and the history of the execution. Of particular concern are assertions that preserve integrity of the objects that are considered constrained data items [CW87].

Constraints are used to enforce specific requirements in a particular type of processing. They incorporate integrity requirements (integrity constraints), separations of duty and fall back rules in the face of potential abort. Fall back rules are necessary in cases where we have a failure and need to restart processing without the intention of undoing all the script's actions until that moment. For instance, in enforcing separation of duty, we may find that a manager who can sign in place of a clerk, needs to undo the actions pertaining to the clerk's role and revert it to a clerk once one is available. The fall back rules would specify the undoing of the manager's action relating to the clerk, have a clerk perform that role before the manager can proceed with the appropriate actions requiring the manager's role. Such cases may require roll forward where the script may follow a completely different execution [WR92].

The following notation will be used in the rest of this chapter:

1. $T = \{t_1, \cdots, t_n\} = \{t_i\}$: a set of transactions.

2. $t_i \rightsquigarrow t_j$: serial execution where t_i executes followed by t_j. Here t_i must commit before t_j starts execution. A serial transaction execution for transactions $t_1 \cdots t_n$ is denoted as $t_1 \rightsquigarrow, \cdots, \rightsquigarrow t_n$.

3. $t_i||t_j$: parallel execution of t_i and t_j. Both t_i and t_j execute independently of each other. A parallel transaction execution for transactions $t_1, \cdots, t_n$ is denoted as $t_1|| \cdots ||t_n$.

4. $\alpha(t_i)$: iterative execution of a transaction t_i. This refers to such cases as when the operation is applied to a set (collection) of objects. For example in a collection type, one can specify iteration of such an operation on the set of object(s) of the type.

Defining a script involves specifying its well-formed transactions, $\{t_1, \cdots, t_n\}$, participating in the script execution and the ordering of the execution. Chunks of the script may be executed serially or in parallel with other chunks. The transactions, $\{t_i\}$, are themselves defined to be well-formed as defined in chapter 2. Clearly, a script from this definition is composed of sub-transactions which can commit or abort individually. In the serial portion of a script, execution proceeds if and only if the preceding portions of the execution have committed.

A script is a complex computation which determines the order of execution, hence the flow of control, of a given set of well-formed transactions. In its simplest form, a script is composed of one transaction (nested or otherwise). It can be composed of several transactions whose order of execution could be serial, parallel, iterative (see above) or a combination of these.

Portions of the script in which transactions execute serially, as in $t_i \rightsquigarrow t_j$, imply that execution control passes from the first transaction to the second when the first transaction completes execution. Hence t_i must complete execution, or commit, before t_j takes over. The serial portion of a script, viewed in this light, defines the dependency of the associated transaction. The effects of a script specified by serial execution hold only if all the transactions in the serial execution commit; otherwise their effects will be compensated for accordingly.

In a parallel execution of transactions, as in $t_i || t_j$, we do not care about the order of execution. Any two or more parallel executions can be executed independently of each other and hence they can be interleaved since a parallel execution can be seen as being concurrent. The effects of a script specified by parallel execution hold only if all the transactions in the execution commit; otherwise their effects will be compensated for accordingly. The difference between a serial and parallel execution is that in the former the commit point comes after the last transaction commits. In the latter, there is a common commit point at which every transaction must commit.

An iterative execution repeats the same set of operations on some given set (collection) of objects (e.g. instances of a type) or some collection object (e.g. an instance of a collection type). For example in set types, iteration repeatedly applies the given operation on the elements of the given set. For instance, if we have a specification of the form $\alpha(t_i)$ in a given type, it means that t_i would be applied repeatedly on elements of some set in that type until the set is exhausted. Since iteration does not specify a dependency between one execution and another, it can be done in parallel, where possible.

Serial, parallel and iterative executions need not involve just one or two scripts. They could be defined to pertain to chunks of executions. For example we can have a script specification as follows:

$$\alpha(t_i \rightsquigarrow t_j \rightsquigarrow (t_k || t_l))$$

Should the script require to abort globally, then we must invoke its compensating script to compensate for actions of the committed portions of the script.

Script execution can be seen as taking the database through a sequence of states based on its execution. A method in a script execution can be viewed as being a transition in database state that takes place upon method execution. For instance, the execution of transactions $\{t_1, \cdots, t_n\}$, starting with Q_0 as the initial state, can be represented as: $Q_0 \vdash_{t_1} Q_1 \vdash_{t_2} \vdash_{t_3} \cdots \vdash_{t_n} Q_n$. The concern is that such transitions cause the database state to change from a consistent state to another consistent state. While the states $Q_0, \cdots, Q_n$ may be valid database states, the document associated

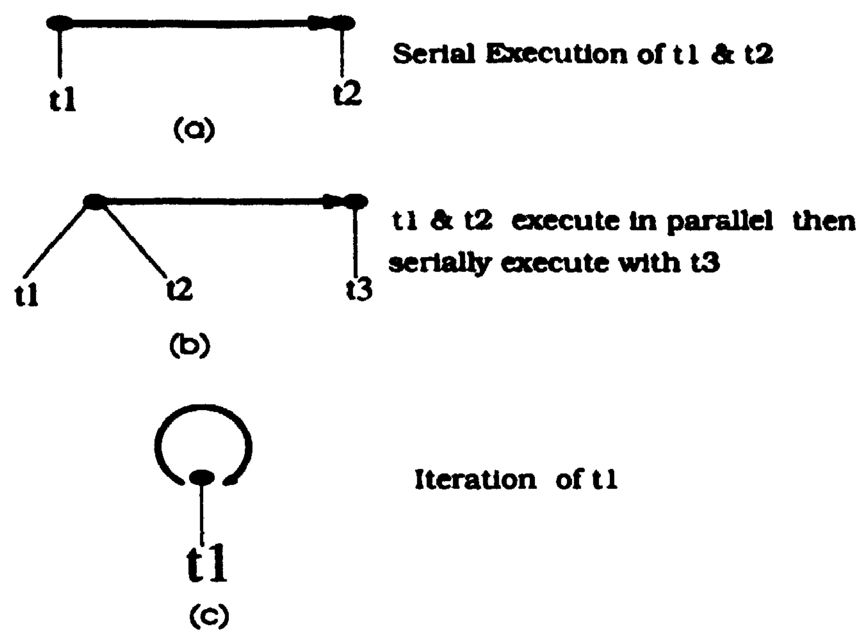
t1
t2
(a)
Serial Execution of t1 & t2
t1
t2
t3
(b)
t1 & t2 execute in parallel then serially execute with t3
t1
(c)
Iteration of t1

Figure 4.2: Depicting Script Execution

with the script will be said to have been successfully processed if state Q_n is attained. Where a script execution aborts, there is the risk of having an invalid document and inconsistent database state. The effect of the compensating transactions (script) is to undo the effects of the aborted script. For example, all debits associated with an incomplete (aborted) cheque will be compensated by corresponding credits, while all credits associated with the same will have corresponding debits.

Compensating transactions restore the input state to its form before the execution of the aborted transaction. Denote this state by Q and the resulting state just before aborting as Q', and let $Q \vdash_{sr_1} Q'$ represent the state transition due to the execution of script sr_1. Then the whole execution with the compensating transaction can be represented as $Q \vdash_{sr_1} Q' \vdash_{sr_1'} Q$, i.e. from an observer's point of view, there is no apparent change in state.

We use *dependencies* to specify the relationship between a script and its compensating transactions (script). In general, a dependency exists between two scrips $sr_i, sr_j \in \mathcal{S}$ denoted $sr_i \leadsto sr_j$, if on commit, sr_i triggers the execution of sr_j. We say sr_j *depends* on sr_i for invocation. For recovery after transaction abort, the dependency relation enables us to specify recovery dependencies when transactions abort, i.e. what compensating transaction is to be invoked when a given script aborts. The effect of this compensating transaction is to undo the effects of the one aborted and can be specified as: **on abort** $sr_i \rightsquigarrow sr'_i$. Considering definition 4.4 we have the restriction that $\forall sr \in \mathcal{SR}, \exists sr' \in \mathcal{SR}' \mid Q_0 \vdash_{sr} Q' \vdash_{sr'} Q_0$ where the relation **on abort** $sr \rightsquigarrow sr'$ is specified.

Unlike [WR92], our compensating transactions are automatically triggered rather than requiring manual intervention. Based on the fact that to date we have saved the context and history of the aborted transaction, it should be possible to enforce the reverse process. Like other transactions, we can define a compensating transaction on compensating transactions. However, we stand the risk of non-termination with such specification. Hence we require that a compensating transaction, on abort, be tried for only a finite number of times or for a finite length of time after which manuai intervention will be necessary. Such manual intervention invocation can be specified using the dependency relation as well.

Compensating actions of a dependent script must undo the effects of the script on which it depends. For instance, the cancellation of a cheque must invalidate all the effects of the voucher that trigg·red its processing. All objects accessed by the voucher must be updated accordingly. In other words if we have *voucher* $\rightsquigarrow$ *cheque* defined with compensating scripts *voucher'* and *cheque'*, respectively, we must have *cheque'* $\rightsquigarrow$ *voucher'*.

Definition 4.5 *A script specification is a script along with the constraints on the script.* □

Given a set of scripts $\mathcal{S}$, a set of methods $\mathcal{M}$ and a set of constrained objects $\mathcal{O}$, we have the relationship: $\mathcal{S} \times \mathcal{O} \times \mathcal{M} \mapsto \mathcal{O}$. A given script execution yields a constrained object that may be subjected to further processing by other scripts.

New scripts can be specified by combining two or more existing scripts. For instance, instead of having the voucher and cheque separately, we can have one cheque-voucher script that uses (possibly with modifications) the existing cheque and voucher scripts.

4.5 Transaction Model Properties

We also need to incorporate the ACID properties as well as their semantics in our model. Unlike the traditional models, we redefine the semantics of these properties to suit the processing in our environment. Atomicity must be managed by means other than traditionally used. Consistency requires that t; ; database be left in some valid state. Isolation, on the other hand, requires considerations of the fact that objects can be complex and locking a whole object may be detrimental to concurrency. Durability requires that a transaction's effects be permanent beyond the life of the execution.

We postulate that the script executions, as presented in this chapter, will preserve the ACID properties of transactions. Our argument stems from the observation that all scripts are composed of transactions and if every transaction preserves the ACID properties, then the overall effect of script execution will preserve these properties.

The following demonstrates that scripts, by their nature, will not violate the ACID properties of transactions from which the scripts are constituted. We take each of the ACID properties in turn.

1. **Atomicity:**

   Each script's actions are considered atomic. An incompletely executed script is allowed to proceed to completion or aborted. Aborting a script, no doubt, requires rollback and compensation of its actions. A committed script generates an authentic legally binding document.[3] An aborted script generates a document that cannot be honoured legally such as a cancelled cheque.

   A script specifies the order of execution in our model. The components of these executions are transactions or transactional executions. Hence, individually, these operations guarantee atomicity. However, the effects of a script are committed if and only if all its components commit. Failure to commit one or more of the transactions of a script will cause the script to abort and trigger compensating operations to undo the effects of those transactions that may have committed already. Thus, extending this argument, we can state the following:

   Conjecture 4.1 *A script which is composed solely of transactions or transactional executions will guarantee atomicity.* □

   To proof this statement it can be argued as follows:

   Proof: Assume that a script does not guarantee atomicity. Then this would imply that there exists some portion of the script which does not guarantee atomicity

[3]Once such a document has been generated, there is no way of undoing its effects within the framework of our model. This can be seen in the same light as a mistakenly cashed cheque. Resolving this situation must resort to other means. We do, however, assure the integrity of the processing as well as the data involved in the processing.

which implies that the portion is not transactional. However, from the script definition, we know that all its components are transactional. Thus we must conclude that our assumption was wrong. It then follows that the script must guarantee atomicity. □

2. **Consistency:**

   Scripts are composed of transactions and transactional executions. Each such execution has a specification that defines its correct execution and guarantees its correctness. Therefore, each of the individual executions preserves consistency. Thus rearranging these executions in any manner (as may be done with the associated operators) must assure database consistency.

3. **Isolation:**

   We do not depend on locking items for isolation. Given our objects can be complex, we can lock only parts of the object on which operation conflict can be anticipated. Moreover, we also use semantic operation information to determine locking (e.g. consecutive increments on a value that is not upper bounded can commute). Thus like the arguments about atomicity and consistency, if all executions within a script are transactional, the script itself must assure isolation.

   We note two types of conflicts, inter- and intra-script conflicts. The former occur within a script while the latter happen between two or more scripts. **Inter-script** conflicts occur when transactions specified within the script execution conflict with each other, notably where there are parallel executions, there is the possibility of such conflict. **Intra-script** conflicts, on the other hand, occur when there is conflict between two scripts as described above. Both types of conflicts must be handled like any transaction conflicts.

   Locking, to facilitate isolation, can be applied to parts of an object where there is likely to be conflict, i.e. where a transaction updates (say) an attribute of an object. For example if we have a cheque that has been initialized with a cheque number, payee, voucher number, etc. and requires signatures for approval, we need not lock this information. However, we must lock the signature attributes as signatures are executed to ensure that not more than one transaction does the update. In undoing the effects of this signature, we may wish to lock all the information associated with the cheque until it has been invalidated.

4. **Durability:**
   Each transaction or transactional execution that compose a script ensures the permanence of its effects when executed. Consequently, when all the transactions commit, the script's effect must be permanent in the database. An aborted script, due to the abort of one or more of the associated transactions, must undo the effects of all those transactions that may have committed via compensating actions. Therefore, a script's effects on the database have a permanent effect on commit.

Next, we must show that ordering the scripts using the serial, parallel and iterative operators does not violate the ACID properties. However, before we demonstrate this, we make the following observations:

1. The transactions execute in an environment that facilitates the preservation of ACID properties of the individual transactions. In other words, the environment provides for locks or timestamps that enforce isolation, commit protocols that assure atomicity, correctness specifications that guarantee consistency, and a means of assuring durability. We do not prescribe the manner of realization of these prope. ies or the nature of mechanisms that realize them. For example, whether a system uses simple or semantics locks, is a matter for that system designer. All that is important in this respect is the guarantee of ACID properties of the individual transactions.

2. From our formulation, every transaction t has a compensating transaction t'. The latter compensates for any effects of the former should the former abort. Given a serial execution of the form $t_i \rightsquigarrow t_j$, th compensation of this script will be done in the order $t_j' \rightsquigarrow t_i'$. For a parallel execution $t_i || t_j$, the compensation is of the order: $t_i' \rightsquigarrow t_j'$.

3. The iteration operator α does not introduce any new transactions to a script. It represents (say) n executions of transaction (say) t_i on some collection object(s). It can be viewed as transaction t_i executing in parallel on different inputs, (e.g. elements of a set object). It can also be regarded as a serial execution in which the input to the operation changes at every invocation, e.g. $t_i \rightsquigarrow t_j \rightsquigarrow \cdots$ for all t_is that constitute its input. Clearly then, in demonstrating the preservation of ACID properties by the operators, we need only worry about the serial and parallel executions.

We make the following claim and demonstrate that script execution of transactions ordered by the three operators introduced so far will always yield transactions. Formally:

Conjecture 4.2 *Let $T = \{t_1, \cdots, t_n\}$ be a set of transactions. Let $\theta = \{\alpha, \rightsquigarrow, ||\}$ (iteration, serial and parallel operators, respectively) be a set of operators which impose an order of execution on two or more $\{t_i\}$s. Then every subset of T, ordered by any subset of θ, will itself be a transaction.* □

Proof: The proof will first show that none of the operators violates transactional properties once it imposed an order of execution of one or more transactions. Next will be to demonstrate the preservation of transactional properties using induction on the number of transactions. Having shown that each of the operators preserves transactional properties of a script composed of one transaction or more transactions, we extend this to n transactions. From here we argue that extending the same argument to $n+1$ transactions does not affect the transactional properties since beyond the n transactions, the extra transaction would have been introduced via one of the operators. Moreover, all we need worry about are the serial or parallel operations since as shown above, the iterative execution can be expressed in terms of either serial or parallel executions. This then will establish the validity of conjecture 4.2.

Let t_i, t_j and t_k be transactions from the set T.

1. t_i, t_j and t_k are transactions and hence, individually, they have transactional properties. The ACID properties hold for the trivial case of a single transaction.

2. $t_1 = t_i \rightsquigarrow t_j$ is a transaction since from the semantics of the serial operator, the effects of t_1 hold if and only if t_i commits to trigger t_j which in turn commits, otherwise the effects of both transactions will be compensated for. Therefore t_1 will be transactional.

3. $t_2 = t_i || t_j$ is a transaction since from the semantics of the parallel operator, the effects of t_2 hold if and only if both t_i and t_j, executing in parallel, commit, otherwise the effects of both transactions will be compensated for. Therefore t_2 will be transactional.

4. Now assume that the ACID properties hold for some script composed of n transactions. This can be regarded as a complex transaction T_n. We then argue that the same properties will hold for $n+1$ transactions.

Let T_{n+1} be the script due to the $n+1$ transactions. The extra transaction in the $n+1$ transactions would have been added into the system ordered either by the serial or parallel operators. Hence T_{n+1} can be expressed either as:

$$T_{n+1} = T_n \rightsquigarrow t_{n+1}$$

or

$$T_{n+1} = T_n || t_{n+1}$$

The former is equivalent to the case shown in item 2 while the latter is the case of item 3.

Conclusion: A set of transactions ordered by the serial, parallel and iterative operators is itself a transaction. □

Therefore script ordering of transactions, using a combination of the serial, parallel and iterative operators as prescribed in this transaction model, preserves the ACID properties of the transactions. Clearly, it follows that scripts in our model are themselves transaction.

4.6 Running Example

This section discusses an example of an environment in which our formulation can be applied. This is an example of an inventory management system that services some manufacturing concern. The inventory management function tracks all of the activities pertaining to the inventory to ensure that it maintains an up to date status of the inventory. It is an integrated inventory management with functions of meeting the parts requirements, reordering to meet desirable inventory levels, accepting deliveries and invoking payments for the deliveries. It is intended to meet requisition requests, generate orders for stock (when necessary) and effect payment for deliveries made. We abstract various processes necessary for realizing the inventory management function. Associated with the processing necessary, we need a means of keeping track of what has happened like the stock requisitioned, the status of the inventory, the type of goods and quantities ordered, etc. We do this using objects (CDIs) which can be updated in accordance with the handling of the CDI type. Hence only authorized users can update (e.g. signing) the object and all such updates will be logged both in the object history and system log.

The first section addresses the procedures necessary in the environment while the next one focuses on how to keep track of the information pertaining to the inventory system.

4.6.1 The Processing

In meeting a requisition, the inventory is updated and if the quantity on hand goes below the reorder levels, an appropriate order is generated and dispatched to an appropriate supplier. The system also processes deliveries from suppliers as well as payment for deliveries made (figure 4.3). In each of the processes, there is a validity (verification) stage at which the authenticity of the process is determined. Validity checking ensures that the process has been initiated by *bona fide* authorized users. For instance, a requisition will have personnel authorized to generate them. This is followed by the execution of the process and finally the signature for the authenticity of the process.

1. **Requisitions Servicing:**
   Requisitions are received from (say) the shop floor showing the parts and quantities required. The requisitions procedure involves ensuring the validity of the requisition, checking whether or not there are sufficient levels to meet the requisition, and servicing the request. Once done, this invokes the inventory update procedure that will ensure the current status is reflected.

2. **Inventory Update:**
   This takes in the quantities of the requisition and updates the associated numbers in the inventory. Updates are intended to make sure that the inventory tracks the current available stock and whatever is on order. An order procedure is generated during the update whenever the reorder point of a given type of stock is reached.

3. **Orders Processing:**
   Whenever it is necessary to make an order, the order processing procedure is invoked. This is done in such a manner that takes into account the consumption rates of the type of stock, the lead time necessary to ensure that stock will be available when needed, and that current stock levels can meet the demand before a delivery is made.

4. **Delivery Processing:**
   The supplier meets a given order by making a delivery of the ordered parts and their associated quantities. These are then entered into the inventory to reflect new status. A completed delivery process could initiate invoicing. As with requisitions, a delivery must trigger an inventory *update*. Deliveries are also associated with the payment (voucher and cheque) process.

5. **Invoice Processing:**
   Invoices are accepted with respect to deliveries made. The invoice process checks for the validity of the voucher like ensuring that there has been a corresponding delivery with respect to a given order. This is essentially a verification process after which it is passed into the payment process.

6. **Payment Processing:**
   This is a two stage process which is triggered by both the invoice and delivery processes. The first stage is the voucher processing task which, when completed, triggers the cheque processing task.

   - **Voucher Processing:**
     Here the deliveries and invoices are determined to be in order following which a voucher is generated . Voucher processing proceeds with approval for payment. A completed voucher process invokes the cheque process.

   - **Cheque Processing:**
     Like all other processes, cheque processing must verify that the voucher(s) and other associated documents are in order before initiating a fresh cheque. This is then signed by duly authorized personnel before being dispatched to the payee (supplier, bank, etc.).

4.6.2 The Objects

In our formulation, we keep track of information pertaining to the above processes using objects. Hence for each type of processing, we have objects that keep the history of a given process. Each object keeps the contractual information pertaining to the process with which it is associated. It keeps track of the quantities, signatures, verifications, etc. that ensure the validity and authenticity of the object. Within each

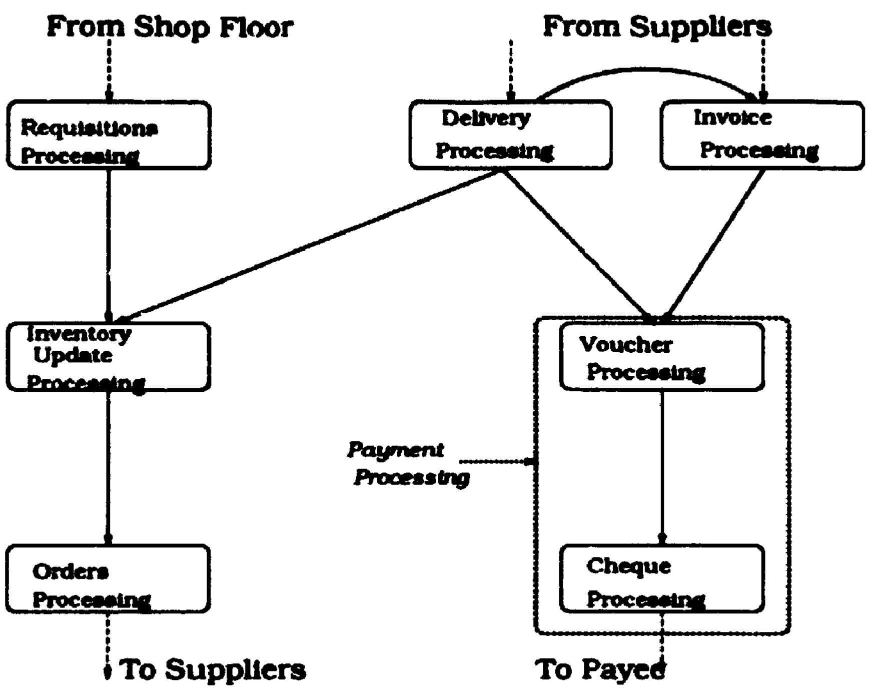

Figure 4.3: Inventory Managen:‹‹ : System Procedures

object, there are associated transactional executions that manipulate its state. The order of execution of these transactions is derived from the associated script.

Each object, in our formulation, is processed based on the script. Whenever a new object is generated, there is an initialization process specific to that object. For instance, in the case of a voucher, it will be initialized with all the relevant information necessary for its intended purpose. The initialization process of any of the objects can be seen as an unfilled blank form to which appropriate approved signatures will be appended. For instance, in tne case of a voucher, information like who the payment is due to, the reason for payment (e.g. with respect to what order and delivery), and signatures confirming that the necessary requirements and approving the payment have been met.

The system must specify which roles (see chapter 5) are authorized to append signatures to what objects where such signature executions will be specified as privileges of that role. There may also be associated relationships among roles authorized for given object transactions which could be subject to the role graph of the next chapter. For example, we can have a case where a manager can play the roles of manager, supervisor as well as clerk while the supervisor can play the role of supervisor and clerk. The clerk, in turn can only execute privileges associated with the clerical role.

Other processing requirements may include procedures that assure that assertions associated with the object of that type, hold. For example in processing a voucher, it would be good to ensure that not only is it a valid payment, but also that the account from which the amount will be drawn has funds. Supplementary processing may involve the request for more allocations should the funds have been depleted below approved levels.

An object processing is considered completed once it is verified that all required processing steps prescribed in the associated script have been accomplished. This may be things like appending appropriate signatures to the document/account that is being processed. Where completion of processing of some object triggers the start of processing of another, control passes from the former to the latter. This is in the case where a *dependency* is defined between two objects' processing. In other words, once one object is completely processed another one must be initiated. For example completion of processing of a voucher can trigger the processing of a related cheque.

We abstract the following:

1. **Requisitions:** Requisition from the inventory is made via a *requisitions note.* This is a document that must be generated and initialed by a clerk, and approved by a project leader and a project manager. Processing the requisition order involves a validation process, confirmation that there is a sufficient stock to meet the requisition, an update of stock levels (where there are sufficient supplies), checking whether a reorder point has been reached and if so, generation of a fresh order for restocking.

   The validation process involves checking that the requisition document is in order, i.e. that all signatures have been appended and that all the appended signatures are valid.

   Confirming whether or not sufficient stock levels exists involves checking that the numbers in current inventory can meet the demand in the requisition. Once this has been confirmed, the requisitioned quantities are used to update the stock.

   Stock level updates involve the new stock levels taking into account the numbers shown in the requisition. New update figures are then checked against reorder levels to determine whether or not to make a fresh order for restocking. To determine the quantities to go in the orders, one can incorporate the rate of consumption and the delivery time to ensure that the orders will be delivered before the stocks become nil.

2. **Orders:** The orders document is generated (triggered) by a check on the stock levels. It requires appending the signatures of the authorized orders clerk and an approval from the officer in charge. This is then transmitted to the supplier (electronically or otherwise). Once processed by the supplier, orders information is used in the processing of an invoice and a delivery note.

3. **Delivery:** Each delivery must be done with respect to some order. The delivery note must bear the order number and information pertaining to the subject that appro.ed the order. Once received an order is used to update the inventory.

4. **Invoice:** Receipt of an invoice triggers the payment process. The invoice is validated both to ensure that an associated order was made and a delivery for the order received. Signatures on the invoice are also checked to determine their validity after which a payment process is initiated.

5. **Voucher:** This is the cheque-voucher example. It is required that a payment, in this context, be made only with respect to a given order and invoice. Receipt of invoice triggers a validation mechanism intended tc verify that the invoice is in order. To be in order, an invoice must have a valid order number, the order and quantities billed must match. Moreover, it must have on its face a valid signature from the supplier.

6. **Cheque:** Once validated, the cheque process is initiated. A voucher is initialized with the invoice numbers, order numbers and any required information before being sent for approval. Approval requires signatures (e.g. two signatures by clerks and one by a supervisor). Once the voucher processing is processed, it triggers the cheque mechanism.

7. **Inventory Log:** The inventory log reflects that current status of the stock in the inventory system at any one time. It shows the current available stock for all types of stock items, the quantities on order, the time of order as well as expected delivery dates. These numbers are updated when (1) a requisition is made for stock (2) an order is generated and when (3) a delivery arrives/accepted.

   The inventory log has an infinite history and hence is handled as an account object.

8. **System Log:** System activities in the system, authorized and unauthorized, must be logged in the system log. The choice of what to keep in the log and

what not to keep depends on what information is seen as necessary for system audit. Like the inventory log, the system log is an account object.

4.6.3 Cheque & Voucher Scripts

Suppose in this example we have the following transactions associated with the cheque and voucher.

V_Init: which initializes the voucher. It generates a copy of the voucher object, allocates it a voucher number, fills information pertaining to the payee ID, associated invoice, the account of budget allocation, associated delivery note, etc.

V_SIG1 and **V_SIG2**: The two signatures required for a voucher to be approved. Once both **V_SIG1** and **V_SIG2** have been executed by duly authorized subjects, the voucher becomes a legally binding document. We may also have a transaction **V_DISP** which dispatches the voucher after it has been processed. We can have the following specification of the script execution:

$$V\_Init \leadsto V\_SIG1 \leadsto V\_SIG2 \leadsto V\_DISP$$

or

$$V\_Init \leadsto (V\_SIG1 || V\_SIG2) \leadsto V\_DISP$$

The former requires that V_SIG1 executes before V_SIG12 while the latter does not care about the order of their execution. In other words, V_SIG1 and V_SIG2 execute in parallel. In each case, V_DISP is invoked to dispatch the voucher.

For cheque processing we can have a similar process in which the following are defined:

C_Init: which initializes the cheque. It generates a copy of the structure (from the type definition) of the cheque object, allocates it a cheque number, fills information pertaining to the payee identity, associated voucher, the account on which it will be drawn, etc.

C_SIG1 and **C_SIG2**: The two signatures required to make the cheque legal, i.e. once both **C_SIG1** and **C_SIG2** have been executed (with any associated constraints such as separation of duty) by authorized subjects, the cheque becomes a legal document. A transaction **C_DISP** takes a duly processed cheque and sends it off (say) to the payee. The script execution can be similar to that of the one for voucher.

Note that the transactions involved could be complex in that they can have built in accesses to data other than that of the associated object. Thus we may have (say) the cheque transactions **C_SIG1** and **C_SIG2** checking that indeed the associated account on which the cheque is drawn, has sufficient funds. They may also check things like the authenticity of the associated voucher authorizations, the delivery note numbers, etc. to ensure that the payment is a valid one before making the cheque legal. Once all this is done, they may log the changes in the system log as well as the associated object histories.

4.7 Summary & Key Contributions

This chapter has presented a transaction model for application in our formulation. We have outlined the requirements envisioned in our applications and explored the shortcomings of the traditional transaction model to meet these requirements which make it unsuitable for our application. This insight formed the basis of our model.

The proposed model recognizes that processing in our application environment is based on prescribed ordered executions of processes of a transactional nature. This gave rise to the proposition for the use of *scripts* which prescribe both the order of execution as well as the compensating transactions for these executions. Scripts are essentially nested transactions in which the nature of nesting is determined by the order and nature of the transactions in the script. Each of these transactions in the script has the ACID properties. Hence in case of abort of a parent transaction, a compensating action undoes the effects of all the children of the aborted transaction.

Since we are operating in a commercial security environment we require that all manipulations of the CDIs be done by TPs and IVPs which must be well-formed transactions. Further, since we are using O-O approaches, the methods must be made transactional and operate based on the principles of TPs. This required the incorporation of these principles in the object model presented in the last chapter.

The concept of scripts used here came from Wacher and Reuter [WR92]. We extended it to incorporate constraints like requiring all object updates to be effected via the script execution. Moreover, we also proposed that the transactions in a script could execute in a serial, parallel or iterative order. These specifications are intended to ensure that all possible computations can be realized via our approach. Transaction nesting is also discussed in [WR92, AA92, WS92, Day93] and [NRZ92]. The concept of finite and infinite histories comes from the recognition that processing environments can be characterized as bounded or unbounded processing. For example documents (cheques, vouchers, etc.) have a finite number of steps in their processing while accounts (audit trails, bank accounts, inventory logs, etc.) are processed as long as a given system lasts.

CHAPTER 5

ROLE-BASED PROTECTION

5.1 Introduction

Roles are implied in the Clark and Wilson model [CW87]. In Lee's [Lee88] words, the Clark and Wilson model essentially reduces to identifying the required roles in a system, determining conflicting roles, ensuring that no user acts in two or more conflicting roles, and letting each specific kind of application data be modified by specific approved transactions acting on behalf of some user executing in an authorized role. The aim of this chapter is to discuss the role concept, provide its formal definition and give an insight into its application in role-based security systems. We shall address the advantages of the role-based approach to system protection and explore its relationship to traditional protection schemes. Consequently, we study the emulation of information flow schemes using role-based protection. Although this thesis focuses on integrity of data objects and operations, we find it prudent to address the issue of secrecy (albeit briefly) when we utilize role-based approaches to realize protection in the traditional sense, such as in mandatory access control. Hence a study of the realization of mandatory access control using the role-based approach is presented.

In section 5.2 we discuss the basis of the role concept, i.e. *implicit authorization.* Roles offer an organizing framework for access rights administration. These rights, administered as privileges, are grouped into roles which are then authorized as single units to users. By defining role relationships and the rules for inferring the access rights based on these relationships, one effectively realizes implicit authorization. In other words, the user's access rights in a system are implicitly determined from the authorized roles, the relationships with other roles in the system, and system information. A user belonging to a group is implicitly authorized to the group's access rights. Moreover, when role inter-relationships are defined to reflect associated functionalities, we can use these relationships to infer access rights that are not explicitly assigned.

In section 5.3 we explain the role concept, its application to security and the advantages of role-based protection. We discuss how roles are used for access rights administration in role-based security systems. A role is a collection of privileges pertaining to system objects and/or resources. It exists separately from both the user and the object or resource it pertains to. A role simply defines what an authorized user, executing in the role, is capable of of doing with system objects and resources. The major advantage of role-based protection is the flexibility with which access rights can be granted and revoked to system users. The key disadvantage of the approach is the complexity of analysis of access rights distribution and the implication of access rights assignment in the system.

Section 5.4 formally defines the term role. The role concept, as used in this thesis, is based on the *privilege* concept. The privilege itself is a *looser* form of the term *capability*. A privilege specifies an object or resource and one or more modes of access. Further, it can be subjected to wide ranging policies in the associated administration. A capability, on the other hand, while specifying an object or resource and the mode of access, must be *unforgeable*; it can be copied as many times as the authorized user may deem necessary. A capability is more specific than a privilege. Viewed this way, a capability is a special case of privilege. We make a distinction between these two and use the privilege concept for role definition. A role, so defined, is a non-empty collection of privileges.

Sections 5.5 and 5.6 present a discussion of role-based protection in the light of traditional protection approaches, namely, information flow analysis and mandatory access control. In section 5.5 we discuss information flow analysis and its application in role-based systems. Section 5.6 presents a realization of mandatory access control using a role-based approach. Section 5.7 offers a summary of this chapter and what we deem to be the key contributions presented here.

5.2 Implicit Authorizations & Roles

The concept of *implicit authorization* [RWBK91, RWK88] is closely related to the use of roles for the administration of access rights in a system. This is a point made by Rabitti et al. [RWBK91, RWK88] who propose a model for authorization for object oriented database systems. The problem of authorization in an object oriented

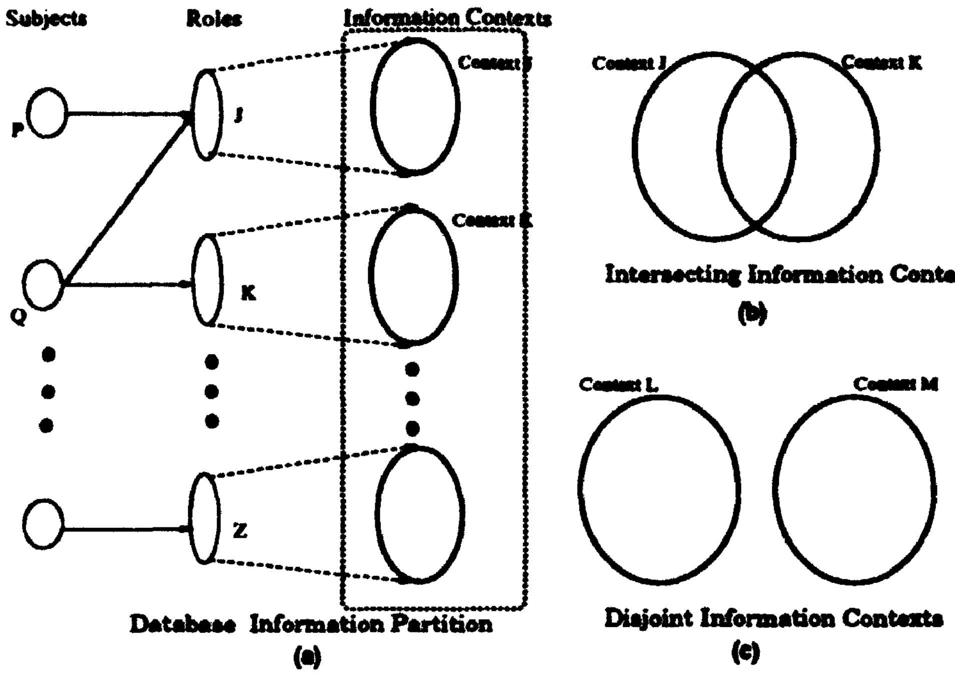

Figure 5.4: Information Partition Via Roles

Notice that this is an inequality relationship, and not equality due to the principle of *aggregation* [DS92]: the total information of a "whole" is greater than or equal to the sum of the information from the individual parts that constitute the whole. For a given role r, its window of information is defined as $INF(r)$.

Consequently, a role-based system can be seen as *partitioning* system information and availing it via the windows defined by the roles, (see figure 5.4). The information available via one such window can be seen as composing a "category" associated with the role. Thus, a role definition can be seen as specifying a "category" (more appropriately, a *role-induced* category) of information represented by its window. Each of these role-induced categories is available to users via user authorization to the role. For purposes of this thesis we shall refer to role-induced categories as contexts; categories will be reserved for system-defined information "classes". A role is said to have an associated context of information determined by its privilege set. Formally:

Definition 5.9 Role Information Context: *A role's information context is that part of system information available via the role.* □

Note that system-defined categories *need not coincide* with role contexts. To determine the context of a given role and its relationship with system-defined categories, we take each privilege and determine its subcontext and how it "straddles" the system-defined ones (figure 5.5). In the most general case these subcontexts straddle

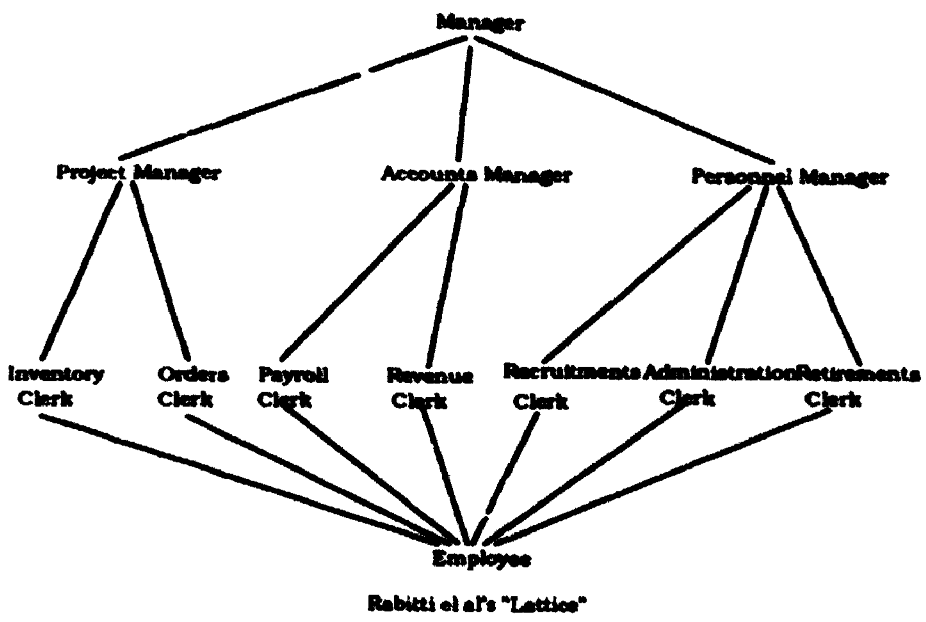

Figure 5.1: An Example of Role Organization

roles. In this way, privilege sharing among related roles in a system is modeled.

This approach to the use of roles offers an organizing framework for system privileges. It reduces the task of specifying user authorization, and with the use of appropriate rules, leaves the task of deducing the authorizations to the system. These are major advantages for both system designers and administrators. For designers, roles present a manner of understanding and testing the implications of their designs. Thus after a design and applying the rules for deduction, designers can understand the implications of their design on the administration of privileges. For system administrators, role based analysis via the applications of rules for implicit deduction, offers a chance to determine individual user and group privileges in the system [TDH92]. It is this organizing framework of roles and the ability to implicitly deduce user privileges that forms the strength of role based protection.

5.3 Roles: Informally

Roles are named groups of related capabilities and privileges pertaining to protection objects or an information system [KM92]. The capabilities and privileges encapsulated in a role are administered as a single unit. Granting a user access to a role authorizes such a user to exercise the capabilities and privileges in the role. In sys-

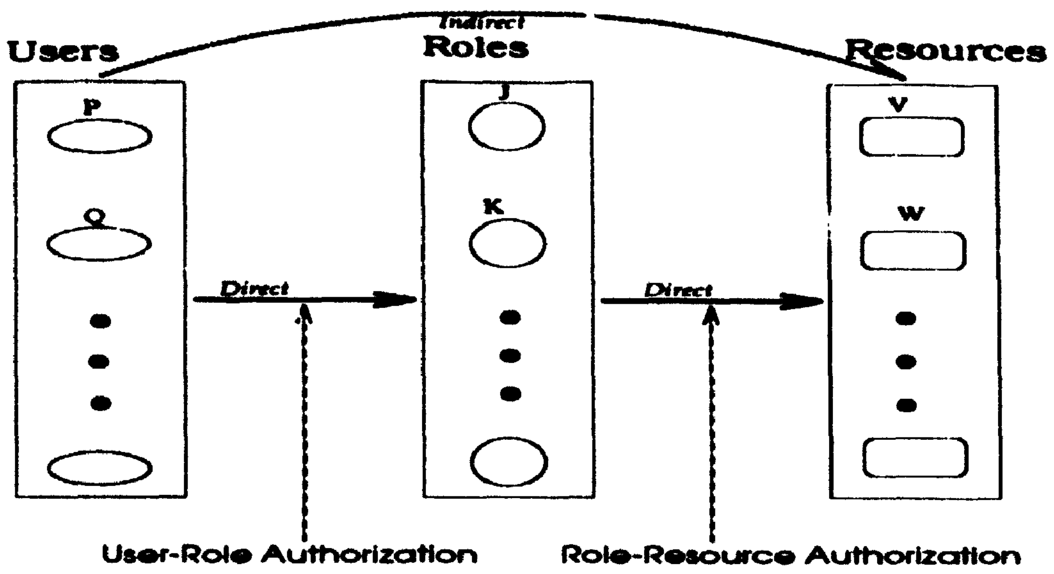

Figure 5.2: User-Role-Resource Relationships

tems where there are a large number of privileges and capabilities, the role approach facilitates their ease of administration. It is more efficient to manage groups of these capabilities and privileges than individual fragments of the same.

Roles exist separately from the users and the objects and/or resources whose access they facilitate (see figure 5.2). In determining user or group privileges in a system, authorization can be given to individual users, groups or roles. Authorization of access for an individual to a role gives such an individual access to all the capabilities and privileges accessible via the role. The same applies to user groups. In effect, roles facilitate the ease of authorization and administration of individual user and group privileges in database systems (see for example [RWBK91, RWK88, TDH92]) for they offer an easy way of assigning and analyzing user capabilities in a system [TDH92]. They simplify the task of administration of system capabilities as well as the ease of management of large numbers of users and user groups with differing ranks (authorities or portfolios) in a system. By organizing users and groups in a suitable manner, this task of system access rights administration is enhanced further.

Depending on the area of application, role authorization can be changed by granting and/or revoking user or user group access to a role dynamically. Role administration is the task of managing roles in the system. It includes defining roles, granting and revoking access privileges to roles for users and user groups, and ordering the roles (where required). Role administration can be a system function or subject to security policies which enforce separation of duty [CW87]. It requires mandatory enforcement; hence it cannot be altered by users other than the system role administrators or system security offi·· (SSO). As we shall show later, even these privileged users must be governed by a set of rules that enforce separatic › of duty if commercial integrity requirements [CW87] have to be met.

A role can be seen as encapsulating the rights, responsibilities, obligations and actions of the role holder. In an organization, the role so specified captures the authority of the role holder and the accountability (liabilities) that goes with wielding such authority. On the other hand, a role has relationships with other roles. For instance, when exercising assigned authority, a subject in an organization must be answerable to another subject acting in another role. These role relationships must be captured by some role organizing structure that expresses these relationships. Role organization to facilitate the ease of access rights administration is the subject of chapter 6.

The main advantage of role-based protection is that it eases the administration of a large number of system privileges. This can be enhanced further should users themselves be grouped such that authorizations to roles are given to user groups instead of individuals. Roles offer flexibility in the granting and revoking of privileges by alteration of a role's privilege list or user/group authorization to the role. Roles implement *least privilege* [BB88, NO93b] ensuring that authorized users access only the information necessary for performing desired tasks. To realize this, a role will include only those privileges necessary to perform associated duties. As well, roles can be designed at the application level which allows for integration of security related application semantics.

The main disadvantage of roles is that the analysis of user privileges and their distribution to various users/user groups can be a very complex process.

5.4 Roles, Capabilities & Privileges: Definitions

The concept of roles is closely related to capabilities [Lin76, BN79]. Capabilities, on the other hand, are subtly related to the concept of *privileges*. In this section we define capabilities, privileges and roles and discuss aspects of roles and privilege management.

Role-based approaches use privileges to realize system protection. A privilege, in this context, determines an object's access rights which can be viewed as a token whose possession confers access rights to the subject possessing it. A privilege is specified by its name and a set of access modes that facilitate access to the associated object. Defined this way, the term privilege carries meaning related to that of a capability. First we define capability [Lin76] as:

Definition 5.1 *<u>Capability:</u> A capability is a pair (x, m) where x refers to an object (or object identifier) and m is a non-empty set of access modes for object x. In is unforgeable and can only be altered by the system security officer.* □

In practice, x can be an object (e.g. object name, OID), object type/class (e.g. in an O-O environment) or attribute name (also in an O-O environment). In systems with simple access modes such as read, write, execute, etc. m is a subset of these access modes. In complex systems, these access modes can be complex where a complex mode of access can be composed of a series of reads, writes and executes. In an O-O environment, this set of access modes is a set of methods. In transactional systems, m would be a set of transactions that manipulate x.

In capability-based protection systems, capability modification, except by the system security officer or except to reduce (e.g. revoke a mode of access) its access rights is not allowed [Lin76]. Capability owners are free to copy and distribute their capabilities as they wish. However, the system has no way of managing the distribution of these capabilities. This is a major shortcoming of capability-based systems.

The desire to enforce policies including those for capability-based systems, creates the need to facilitate modifications other than those allowed in capability-based systems. For instance where discretionary access control (DAC) is enforced, administration of access rights associated with a certain object may be the task of that object's *"owner"*, while in mandatory access control (MAC) systems, only the system security officer (SSO) may modify access rights. This need leads to the concept we term *privilege* which, for purposes of this thesis, is defined as:

Definition 5.2 *<u>Privilege:</u> A privilege is a pair (x, m) where x refers to an object (or object identifier) and m is a non-empty set of access modes for object x.* □

A given privilege definition can be seen as a computation pertaining to the specified object x and access modes m. Depending on the associated set of access modes, a privilege execution can cause change, reveal or add to system information. It can cause the creation or deletion of objects. It can cause the granting of other privileges, creation or deletion of roles, etc. Given that the exact nature of the effects of a given privilege will depend on the desired application, we shall leave the exact details for the application designer. Since privileges are intended for security administration, the security policy must specify how they are administered. In our case, the initialization and modification of a privilege must be authorized. In general, privilege modification will be defined in the form of privileges. Authorization to the execution of such privileges realize such modification. We designate the universal set of privileges in a system by $\mathcal{PV}$.

From definitions 5.1 and 5.2 it is clear that a capability is a special case of a privilege. A capability has a more restrictive security policy specific to capability based systems. Conversely, a privilege is a more general form of capability; it can be specialized to a wide range of security policies as may be necessary in any given system.

There is a *major* difference between privileges and capabilities, however subtle it may appear on the surface. In capability-based protection systems, the term capability connotes some *token* whose possession facilitates access to the object named in the capability in the access mode specified. Capabilities are taken to be resilient to attempts of forgery hence remain unforgeable once specified and assigned. In capability-based systems, access rights administration via capability administration (including assignment, revocation and modification) and definition is a system function; although capability owners can copy and distribute their capabilities as they wish, they cannot alter them.

A privilege, on the other hand, can be seen as a capability when it conforms to the requirements of the capability definition. However, privilege definition and administration can be subjected to a variety of policies. Such policies can range from *ad hoc* ones to formal ones like those for capability specification and administration. Indeed, even the form of privileges can range from *ad hoc* to formal. Capability-based policies are specific in that capability administration is a system function. Whereas capability owners can copy and distribute the capability to any extent they desire, they cannot alter it. Privilege administration, on the other hand, will depend on the policies of the application. For instance where it conforms to capability definition, privilege administration will be a system function. In other cases, as in commercial security, privilege administration can itself be defined as a privilege or set of privileges. The choice of this form of privilege definition is deliberate in that we wish to leave the specific policies pertaining to its specification to be determined by system specific requirements.

Privileges/capabilities in a system facilitate system resource access which allows the performance of some function in the system. It is, at times, expedient to collect some functions and assign the collection as a single unit. This collection of privileges is what we term a *role* which we formally define as:

Definition 5.3 *Role: A role is a named collection of privileges. It is a pair (rname,rpset) where rname is the role name and rpset is the privilege set.* □

For a given role **r**, **r.rname** and **r.rpset** refer to the name and privilege set of **r**, respectively. From a computational point of view, a role specifies the set of computations (privileges) possible via authorization to the role. The effects of invoking these computations would, at the least, be equal to the union of the effects caused by the individual privileges in the role's privilege set. We designate the universal set of roles in a system by $\mathcal{R}$.

A role, based on the privileges associated with it, exhibits a certain behaviour. The role behaviour, on the other hand, is determined by its set of privileges. Since it is intended that roles' behaviour be unique in a system, no two roles should have identical behaviour. Consequently, no two roles may have identical privilege sets.

Defined this way, a role captures functionality achievable via the role, what Dobson and McDermid [DM89] term a *functional role*. Roles can also be related to each other if they have overlapping (shared) privileges. This aspect of role definition is termed *structural;* roles defined this way are referred to as *structural roles* [DM89]. As we shall see in section 6.2, there are a number of structures (hierarchies [TDH92], lattices [RWK88, RWBK91, NO93b], Ntrees [San88, San89]) that can be utilized to capture the structural aspects of role definition.

Since roles are referred to by name, the **rnames** must be unique in a system while the **rpset** can contain privileges relating to arbitrary objects, types, attributes or indeed any system resource. The privilege set, **rpset**, associated with a role, is a set-valued entity consisting of a set (ordered or otherwise) of privileges at the disposal of the role in question. It can be empty. A role's name, **rname**, serves as its identity and given such a name, it must be possible to completely determine its associated privileges.[2]

Ordering, of one kind or the other for privilege execution, is useful in determining the processing steps executed in a particular role. Considering the role as a "processing station" for some stage in some processing sequence, explains the picture more clearly. In this respect, execution in the role can be viewed as having specific procedures that must be adhered to for correct processing. For instance, in an office

[2]For the moment we refer to those privileges directly specified in the role. However, as will be seen later, with defined relationships among roles, a role can have privileges associated indirectly.

environment, such procedures may include the entry of a receipt (e.g. the origin of the current receipt, document serial number, etc.) of the document being processed into some log, actual processing and dispatch entry (e.g. dispatch by whom and to what destination) into a dispatch log for some instance of the processing. Procedures such as communication with the next stage in the processing can be built into these routines and may include digital signatures, if required. In our formulation, the privilege set determines these procedures. The order of their execution in the related role is determined by the associated script defined in the class of the associated object.

In our model, we assume generalized access modes. Procedures associated with a given privilege set can be simple (such as reads/writes/etc.) or complex (e.g. method invocation sequence or transactional invocations). Method and transactional invocations will adhere to the method and transactional invocation paradigms outlined in chapters 3 and 4, respectively. Moreover, transactional executions can be defined in terms of scripts, where such processing requirement exists.

Authorization to a role puts the role's privileges at the disposal of the authorized subject and thus confers the access rights (functionality [DM89]) encapsulated in the role to the subject. Role authorization can be a system function (as in MAC) or can be based on role *ownership* (in the case of DAC).

5.5 Roles & Information Flow

This section discusses role-based security from the point of view of *information flow.* We apply the concepts of information flow to formulate rules for secure role-based security systems. Information flow analysis relies on the operations on data. Information flows from one data object into another when a change in the first object causes a change in the second one. An operation that causes such a change is said to cause information flow.

Information flow analysis techniques have been applied in the design and analysis of secure computer systems (see for example [Den76]). These techniques rely on operations on objects and the changes caused in other objects. In order to apply information flow analysis techniques for security systems, we must specify categories of information that exist in the system; no other categories[3], other than those specified, exist. Further, we assign each object (objects are the information bearing entities) to

[3]Most literature uses the term *classes*. We opt to use the term *category* to avoid potential confusion with the term class in O-O modeling.

at least one of these legal categories; every object must belong to some legal category. A specification of allowable information flows between the categories is made. This forms the basis of determination of a security violation; to preserve security in such a system, no information flow, other than those specified by the *legal flows* should be allowed. Consequently, every operation that causes information to flow from one category to another, must be guaranteed to ensure that such a flow is legal. An illegal flow is one that is not consistent with the security policy. A violation of security occurs when there is an illegal information flow.

Categories (traditionally referred to as classes) of information specify the legal "groupings" into which system objects belong [Den76]. A category can be seen as a "container" of information, with its information held in the objects therein. Information is allowed to flow freely within the category hence no distinction, in terms of information, is made between the information held in objects in the same category.

The effect of defining roles in a system can be seen as inducing information *contexts*. A context of information thus pertains to that part of system information accessible via a given role. Such a context can straddle more than one system category. Further, defining a role scheme can cause information flows in a given system. These information flows, termed role-induced flows, can occur between different system-defined categories. In this section we will discuss the manner in which role definition interacts with system categories, potential role-induced flows and how they impact on system security.

5.5.1 Information Flow Basics

Information flow analysis relies on operations and their effects on data. Information is said to flow from one data object into another when a change in the first object causes a change in the second one. An operation that causes such a change is said to cause information flow. Information flow analysis focuses on the inputs to operations and the outputs of the operations to determine whether information in the inputs finds its way to the outputs. The information flow problem can be stated thus [McH85]:

Problem 5.1 The information Flow Problem: *Given a program and its sets of input and output variables, determine for each output variable, the subset of the input variable set about which it might contain information after execution of the program.* □

Information is said to flow from the subset of inputs to the outputs which contain information about this subset of inputs. Information flows can be explicit (as in assignment) or implicit (as in conditional statements) [Liu80]. They can be due to functional dependency (e.g. x depends on y, hence there is a flow from y to x) or deductive (if knowing the value of x implies knowing the value of y, hence there is a flow from y to x) [Mil81]. In general, information flows from x to y (denoted $x \longrightarrow y$) when information stored in x flows to y or when it is used to derive information about y [Liu80]. Implicit in this statement is the fact that, for there to be an information flow, an input causes change in the output to which information flows. We have the following information flow axiom [DD77, Liu80]:

Axiom 5.1 *Basic Information Flow axiom: A flow $x \longrightarrow y$ occurs only when the value of y is updated.* □

5.5.2 Information Flow & Security

Information flow analysis has been applied in the determination of the security of information in a given system (see for example [Den76, DD77, Liu80, Mil81, McH85, Fol87, Fol91]). By analysing the flow of information occasioned by the execution of some operation, it is possible to determine whether such a flow violates a given security policy. Axiom 5.1 forms the basis of such analysis. In a given system, the security policy will specify which flows are legal. Any flows, other than those specified, would violate security. Legal flows can be seen as authorized flows while illegal flows are unauthorized. In general, the security policy specifies categories of information into which system objects belong and the permitted information flows between the categories. A category can be seen as a "cluster" of information within which there is no restriction of information flow among the objects.

Definition 5.4 *Category: A category is a system-defined "class" of information within which information can flow freely between objects.* □

As part of such a security policy, we must specify the following:

1. The flow rules. In other words, what determines when there is an information flow? As seen earlier, information flows from an input to an output when a change in the input causes a change in the output. In this respect the basic information flow axiom (see axiom 5.1) would form the basis of such flow rules.

2. The legal categories into which objects in the system are assigned. These comprise the legal system information categories. No other categories will exist outside of those specified here.

3. Which objects belong to what categories? Each object must belong to *at least* one such category and no object can belong to any other category other than those specified in item 2. Object category assignment must not violate the desired flow policy.

4. The flow policy: what are the *legal* flows in the system? In other words, from what category and into what category should information flow? Flows that are explicitly specified in the flow policy are said to be ***direct flows***. Direct flows can result in ***indirect*** or ***implicit*** flows, i.e. those flows that can be inferred from the direct flows. This is due to the fact that information flow is transitive. Hence given three information categories X, Y and Z, if $X \longrightarrow Y$ and $Y \longrightarrow Z$, it follows that information flows from X to Z, i.e. $X \longrightarrow Z$ [Den76, Liu80].

   Axiom 5.2 Information Flow Transitivity Axiom: *Information flow is transitive, i.e. given that* $X \longrightarrow Y \wedge Y \longrightarrow Z \Rightarrow X \longrightarrow Z$. □

   Specification of legal flows of information includes aspects of combining information from two or more categories. A flow policy will specify into which category such combined information will flow. In Denning's lattice model [Den76], this involves specifying the ***least upper bound*** (lub) for any two or more categories. Similarly, the ***greatest lower bound*** (glb) for two or more categories depicts the category that is the common source of information for the categories. The legal flows in the system are those that are explicitly defined in the flow policy and those resulting from the explicit specification of flows based on the transitivity axiom. A security violation occurs when there is an illegal information flow. Formally:

   Definition 5.5 Security Violation: *In a system with a specified set of legal information flows, a security violation occurs when there is information flow other than the legal flows.* □

Two things imply a legal information flow: (1) the information flow between objects of the same category or (2) information flow between objects of different categories but which flows are permitted by the security policy. Any other flow is illegal.

5.5.3 Information Flow: Policy & Implementation Consistency

In practice, the *information flow policy* of section 5.5.2 would form part of an overall system security policy which any given realization must adhere to. An implementation would aim at ensuring that all flows in the implementation are according to the stated policy. Any flow in the implementation without a corresponding flow due to the flow policy can be seen as *illegal*. Such a flow causes a security violation (definition 5.5). It can be seen as *unauthorized*. A secure system must ensure that any such flows confine information to its category. This is the basis of the *confinement problem* [McH85, Liu80]:

Problem 5.2 The Confinement Problem: *Given a system with categories of authorization, a confinement problem exists where there are inputs and outputs in two or more categories of authorization. A problem occurs when information flows from input in one category to output in another category.* □

The *confinement problem* merely recognizes that there can be information flow from one category into another. Such a flow need not be a security violation. A security violation occurs when there is a confinement problem where the resulting flow contravenes the flow policy, i.e. there is confinement problem due to an illegal information flow. Stated differently, "a program is secure if no execution of it can result in an illegal information flow from some input x to another output y" [Liu80].

Items 1, 2, 3 and 4 above relating to flow policy and the confinement problem (problem 5.2) can form the basis for specification of an information flow scheme that ensures system security.

Let a system have information categories $\mathcal{CO}$, associated operations $\mathcal{OP}$ and some information flow scheme $\mathcal{F}$ based on some security policy $\mathcal{P}$. Let the set of flows $\mathcal{F}(\mathcal{P})$ be those flows resulting from the flow scheme and the policy $\mathcal{P}$. Clearly, $\mathcal{F}(\mathcal{P})$ contains both the explicit and implicit flows derived from the flow scheme and policy. A function f enumerates all such flows, i.e. $f : \mathcal{P} \times 2^{\mathcal{CO}} \times \mathcal{F} \times 2^{\mathcal{OP}} \mapsto \mathcal{F}(\mathcal{P})$. $\mathcal{F}(\mathcal{P})$ is the set of all *legal* information flows.

An implementation, $\mathcal{I}$, would result in a set of information flows which may or may not be equivalent (see definition 5.6) to $\mathcal{F}(\mathcal{P})$. Let such a set of flows be $\mathcal{F}(\mathcal{I})$ which is the result of some subset of the operations and categories. Let the function ϕ enumerate all such flows: $\phi : \mathcal{P} \times 2^{\mathcal{CO}} \times \mathcal{I} \times 2^{\mathcal{OP}} \mapsto \mathcal{F}(\mathcal{I})$. $\mathcal{F}(\mathcal{I})$ is the set of actual flows in the system.

Denote the categories specified in the security policy and flow scheme by $\mathcal{CO}_{\mathcal{P}}$. Define $\mathcal{CO}_{\mathcal{I}}$ similarly for the implementation $\mathcal{I}$.

Definition 5.6 *Information Flow Set Equivalence: Two sets of information flows, $\mathcal{F}(\mathcal{P})$ and $\mathcal{F}(\mathcal{I})$ are equivalent, i.e. $\mathcal{F}(\mathcal{P}) \equiv \mathcal{F}(\mathcal{I})$, if and only if (1) they both deal with the same set of categories of information, i.e. $\mathcal{CO}_{\mathcal{P}} = \mathcal{CO}_{\mathcal{I}}$ and (2) for every flow in $\mathcal{F}(\mathcal{P})$ there is an equivalent flow in $\mathcal{F}(\mathcal{I})$ and vice versa.* □

Where $\mathcal{F}(\mathcal{P})$ and $\mathcal{F}(\mathcal{I})$ are equivalent, the information flows due to the implementation, $\mathcal{I}$, are exactly those specified resulting from the system security policy and the given flow scheme. A system in which $\mathcal{F}(\mathcal{P})$ and $\mathcal{F}(\mathcal{I})$ are equivalent, is secure; no flow due to the implementation contravenes any of the flows due to the security policy and flow scheme.

It is possible to designate the *flow set equivalence* as the criterion for determining whether a given implementation $\mathcal{I}$ is secure with respect to some security policy $\mathcal{P}$. However, such a criterion is unnecessarily more strict than may be desirable for the reason that the implementation *need not* realize all legal information flows. It is sufficient that flows due to the implementation do not violate any of those resulting from the flow policy and flow scheme. Where a flow set due to an implementation does not violate any of those specified, we say the the former is *consistent* with the latter.

Definition 5.7 *Information Flow Set Consistency: A set of information flows, $\mathcal{F}(\mathcal{I})$, is consistent with another set of information flows, $\mathcal{F}(\mathcal{P})$, if and only if (1) $\mathcal{F}(\mathcal{I})$'s categories are a subset of or equivalent $\mathcal{F}(\mathcal{P})$'s categories, i.e. $\mathcal{CO}_{\mathcal{I}} \subseteq \mathcal{CO}_{\mathcal{P}}$ and (2) for every flow in $\mathcal{F}(\mathcal{I})$ there is an equivalent flow in $\mathcal{F}(\mathcal{P})$. In other words, $\mathcal{F}(\mathcal{I})$ is consistent with $\mathcal{F}(\mathcal{P})$ if $\mathcal{CO}_{\mathcal{I}} \subseteq \mathcal{CO}_{\mathcal{P}}$ and $\mathcal{F}(\mathcal{I}) \subseteq \mathcal{F}(\mathcal{P})$.* □

In general, given a policy $\mathcal{P}$ and an implementation $\mathcal{I}$, we say that the implementation is *flow consistent* with the policy if and only if $\mathcal{F}(\mathcal{I}) \subseteq \mathcal{F}(\mathcal{P})$. This implies that *equivalence* is a special case of *consistency*.

Flow consistency is important for system security. System security in this case is determined by two key issues: (1) implementation flow consistency with respect to the stated security policy and (2) user authorization to the implemented operations. Clearly, where the implementation is flow consistent with the policy and the a user executing some legal operation is authorized, the system is secure. Consequently, a system is secure if and only if all operations are executed by authorized users and the implementation is consistent with the system security policy. Flow consistency alone does not guarantee system security.

Constraint 5.1 *Information Consistency Constraint: A system is secure, with respect to some security policy $\mathcal{P}$, if the flows $\mathcal{F}(\mathcal{I})$ resulting from the system implementation are consistent with the flows, $\mathcal{F}(\mathcal{P})$, specified by the policy.* □

Constraint 5.1 forms the ***key*** criterion for assessing the security of an implementation $\mathcal{I}$ with respect to a given policy $\mathcal{P}$. Given a policy $\mathcal{P}$ and an implementation $\mathcal{I}$, a natural question to ask is whether $\mathcal{I}$ is consistent with $\mathcal{P}$. In other words, is the implementation secure with respect to the policy? Formally:

Problem 5.3 *Policy & Implementation Consistency Problem: Given a security policy $\mathcal{P}$ and an implementation $\mathcal{I}$, is $\mathcal{I}$ secure with respect to $\mathcal{P}$?* □

To answer this question, we consider the information flows due to the security policy and those due to the implementation. We then construct respective graphs of both. To determine consistency, we use the ***subgraph isomorphism problem***.

A subgraph of a graph $G = (V, E)$ (where V represents the vertices and E the edges) is defined as a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$ [CLR90, Baa88, Man89]. A set of flows $\mathcal{F}(\mathcal{P})$ can be represented by a directed graph $\mathcal{F}(\mathcal{P}) = (\mathcal{CO}, \mathcal{CF})$ where $\mathcal{CO}$ is a set of categories (the nodes) and $\mathcal{CF}$ is a set of edges connecting the categories. Information flows from one category co_i to category co_j if and only if we have directed edge $\langle co_i, co_j\rangle \in \mathcal{CF}$. Similarly, $\mathcal{F}(\mathcal{I}) = (\mathcal{CO}_\mathcal{I}, \mathcal{CF}_\mathcal{I})$. Consequently, $\mathcal{F}(\mathcal{I})$ is consistent with $\mathcal{F}(\mathcal{P})$ if and only if, in their respective graphs, $\mathcal{CO}_\mathcal{I} \subseteq \mathcal{CO}$ and $\mathcal{CF}_\mathcal{I} \subseteq \mathcal{CF}$. Clearly, with this condition, the graph of $\mathcal{F}(\mathcal{I})$ must be a subgraph of the graph of $\mathcal{F}(\mathcal{P})$.

Example 5.1 Consider a system with categories A, B, C, D, E and F and a flow scheme A$\longrightarrow$B, B$\longrightarrow$ C, D$\longrightarrow$E and D$\longrightarrow$F. The set of legal flows is $\mathcal{F}(\mathcal{P})$={ A$\longrightarrow$B, A$\longrightarrow$C, B$\longrightarrow$C, D$\longrightarrow$E, D$\longrightarrow$F}. Let some implementation result in flows A$\longrightarrow$B, A$\longrightarrow$C, and D$\longrightarrow$F, hence $\mathcal{F}(\mathcal{I})$={A$\longrightarrow$B, A$\longrightarrow$C, D$\longrightarrow$F}. Clearly, $\mathcal{F}(\mathcal{I}) \subseteq \mathcal{F}(\mathcal{P})$. If only authorized users invoke operations in the system and where the operations execute correctly, then the system will be secure. □

Determining whether a given flow set is equivalent to another, in the general case, is not an easy problem as demonstrated by the following lemma.

Lemma 5.1 *Given two sets of information flows $\mathcal{F}(\mathcal{I})$ and $\mathcal{F}(\mathcal{P})$, $\mathcal{F}(\mathcal{I})$ is consistent with $\mathcal{F}(\mathcal{P})$ if and only if $\mathcal{F}(\mathcal{I})$ is a subgraph of $\mathcal{F}(\mathcal{P})$.* □

In the general case, determining whether $\mathcal{F}(\mathcal{I})$ is consistent with $\mathcal{F}(\mathcal{P})$ is NP-complete. This follows from the reasoning that determination of $\mathcal{F}(\mathcal{I})$'s consistency with $\mathcal{F}(\mathcal{P})$ is to answer the question whether $\mathcal{F}(\mathcal{I})$ is a subgraph of $\mathcal{F}(\mathcal{P})$. This is the **subgraph isomorphism problem** [CLR90, Baa88, Man89]. We know that this problem is NP-complete [CLR90], hence the consistency problem is NP-complete, in the general case.

However, determining flow consistency is an instance of the subgraph isomorphism problem in which the non-determinism has been resolved because the node mapping is known. Category labels are the same in both cases. What remains is to match the given nodes and edges of the two given flows. This can be done in polynomial time $O(n^2)$ where n is the number of nodes in the graph.

We use matrix representation to solve this problem. A given set of flows $\mathcal{F}(\mathcal{X})$ can be represented by an $n \times n$ matrix M in which $M[i,j] = 1$ whenever there is a directed edge (co_i, co_j). The entry is 0, otherwise. Let matrices $M_{\mathcal{P}}$ and $M_{\mathcal{I}}$ be of the form of M and represent flows $\mathcal{F}(\mathcal{P})$ and $\mathcal{F}(\mathcal{I})$, respectively. We can answer whether $\mathcal{F}(\mathcal{I}) \subseteq \mathcal{F}(\mathcal{P})$ by checking that whenever $M_{\mathcal{I}}$ has a 1 entry, there is a corresponding 1 entry in $M_{\mathcal{P}}$. We say $\mathcal{F}(\mathcal{I})$ is inconsistent with $\mathcal{F}(\mathcal{P})$ if there at least one 1 entry in $M_{\mathcal{I}}$ without a match in $M_{\mathcal{P}}$. There is consistency if every 1 entry in the former has a corresponding 1 entry in the latter.

5.5.4 Roles & Information Flow

A role facilitates access to information via its given set of privileges. Such information can be seen as a *context* available to users authorized to the role. This section examines how these contexts arise by revisiting the privilege definition, and defines a context as the union of the *subcontexts* of information accessible via each of the role's privileges. We then address the effect of these contexts on system-defined categories and propose a methodology for determining whether a given role-based scheme *violates* system security.

5.5.4.1 Privileges & Information Flow

A privilege has been defined as a pair (x, m) with x being an object identifier and m a set of *valid* access modes for x (definition 5.2). This definition prescribes that the privilege execution will be guaranteed to get its input from named sources [GMP92], in this case the object x and any of the associated parameters of the access modes.

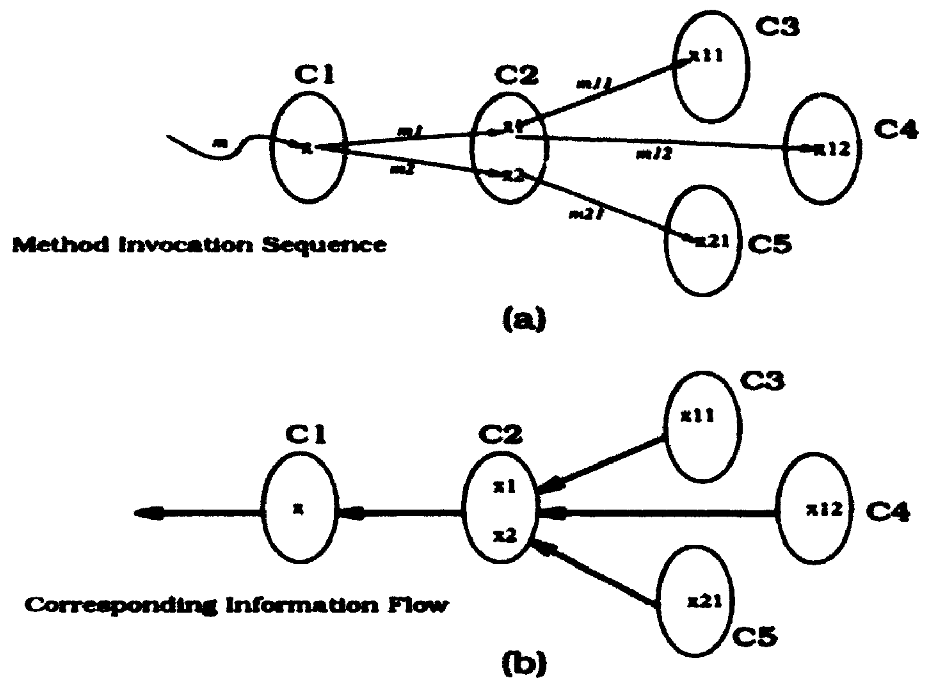

Figure 5.3: Method Invocation Information Flow

This execution facilitates a *potential* transformation. It causes some information to be available via the execution and may cause information to flow in the process. Moreover, since the execution pertains to system information, it offers a form of access to system information. Each privilege can thus be seen as a *subcontext* of information access. The choice of the term subcontext is deliberate since even when information is accessible via such a privilege, it must be accessed within some role.

Definition 5.8 *Privilege Information Subcontext: Privilege information subcontext is that piece of system information accessible via a given privilege.* □

To explain this further, we consider O-O methods as the basis of privilege definition. Let x be the object identifier and m be a set of methods valid for x. The methods may have parameters other than x and their execution may invoke other methods or create new objects. In general, a method invocation sequence can be represented by a tree (figures 3.2 and 5.3a) [JK90]. In figure 5.3 the m_is represent method invocations, the $C1, \cdots$, are system-defined categories (those involved in the invocation) while the x_is stand for objects associated with a method invocation. The arrows in figure 5.3a indicate those invocations which cross categories/objects while those in figure 5.3b show the direction of information flow occasioned by the invocations.

Such invocations can facilitate access to information in the different categories involved in the invocations. The information accessible from a category via an invocation will be no greater than the information contained in the category. An invocation such as that of figure 5.3 yields access to more that one category and hence a fragment of information from each of the categories in question. A subcontext of an invocation is the aggregate of all information fragments from the individual categories involved in the invocation. In our case, the aggregate of information fragments accessed in categories $C1, \cdots, C5$ constitute the subcontext of m. These fragments of information, in turn, are associated with the objects involved in the invocation. Therefore, the subcontext of m would contain the fragments of information pertaining to the x_is in the figure.

The method in this case can be seen as a mini-window into system information pertaining to all the objects "touched" by the invocation. It is the information available via this mini-window that we call a subcontext for the given privilege. The associated *potential* information flow is depicted in figure 5.3b. Note that we call this potential flows because we assumed that all invocations cause some changes. Hence in the case that all such invocations are associated with changes, we shall expect these flows. In practice, however, some invocations may not cause any changes and hence may not cause information flow (see axiom 5.1).

5.5.4.2 Roles & Information Contexts

A role facilitates access to a given set of objects using the specified modes of access in the associated privilege set. Seen this way a role acts as a *window* to system information. Information available via the role window is determined by the role's privilege set. It is, at the *least*, the union of all the information available via the individual privileges in its privilege set.

Given some role $r \in \mathcal{R}$ with an associated privilege set *r.rpset*, let $INF(t)$ represent the "quantity" (of some measure of information) of information accessible via some role or privilege t. Where some pv is in some role r's privilege set, it is true that:

$$INF(pv) \leq INF(r)$$

In other words, the measure of information in a privilege cannot exceed that in its associated role. Yet, the "sum" of all information in a role's privileges is *always* less than or equal to the information available via the role. It follows that:

$$INF(r) \geq \bigcup_{pv \in r.rpset} INF(pv)$$

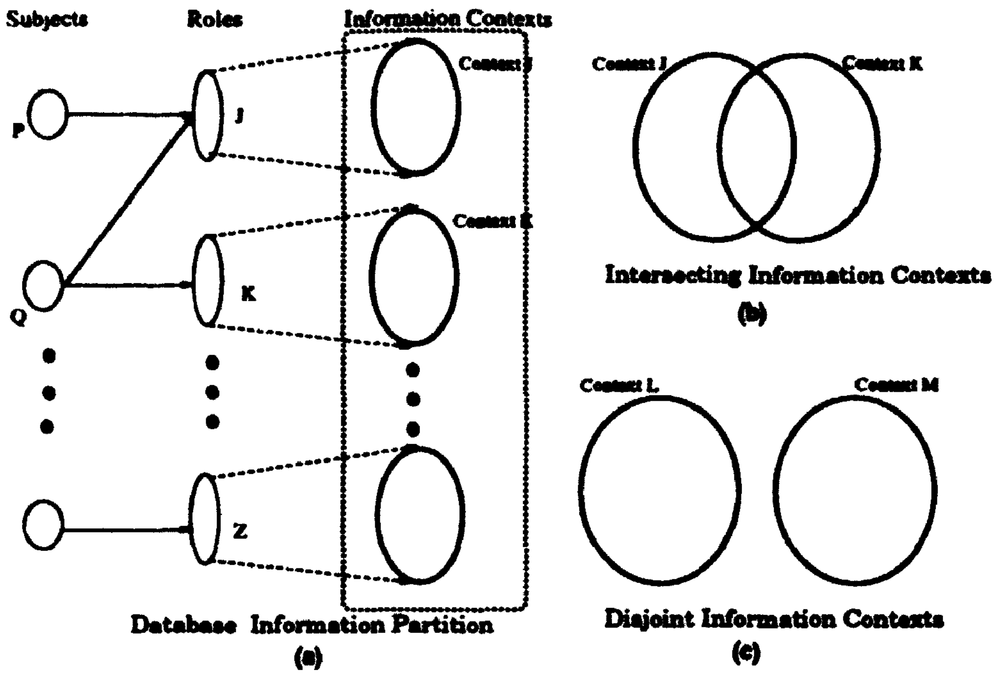

Figure 5.4: Information Partition Via Roles

Notice that this is an inequality relationship, and not equality due to the principle of *aggregation* [DS92]: the total information of a "whole" is greater than or equal to the sum of the information from the individual parts that constitute the whole. For a given role r, its window of information is defined as $INF(r)$.

Consequently, a role-based system can be seen as *partitioning* system information and availing it via the windows defined by the roles, (see figure 5.4). The information available via one such window can be seen as composing a "category" associated with the role. Thus, a role definition can be seen as specifying a "category" (more appropriately, a *role-induced* category) of information represented by its window. Each of these role-induced categories is available to users via user authorization to the role. For purposes of this thesis we shall refer to role-induced categories as contexts; categories will be reserved for system-defined information "classes". A role is said to have an associated context of information determined by its privilege set. Formally:

Definition 5.9 Role Information Context: *A role's information context is that part of system information available via the role.* □

Note that system-defined categories *need not coincide* with role contexts. To determine the context of a given role and its relationship with system-defined categories, we take each privilege and determine its subcontext and how it "straddles" the system-defined ones (figure 5.5). In the most general case these subcontexts straddle

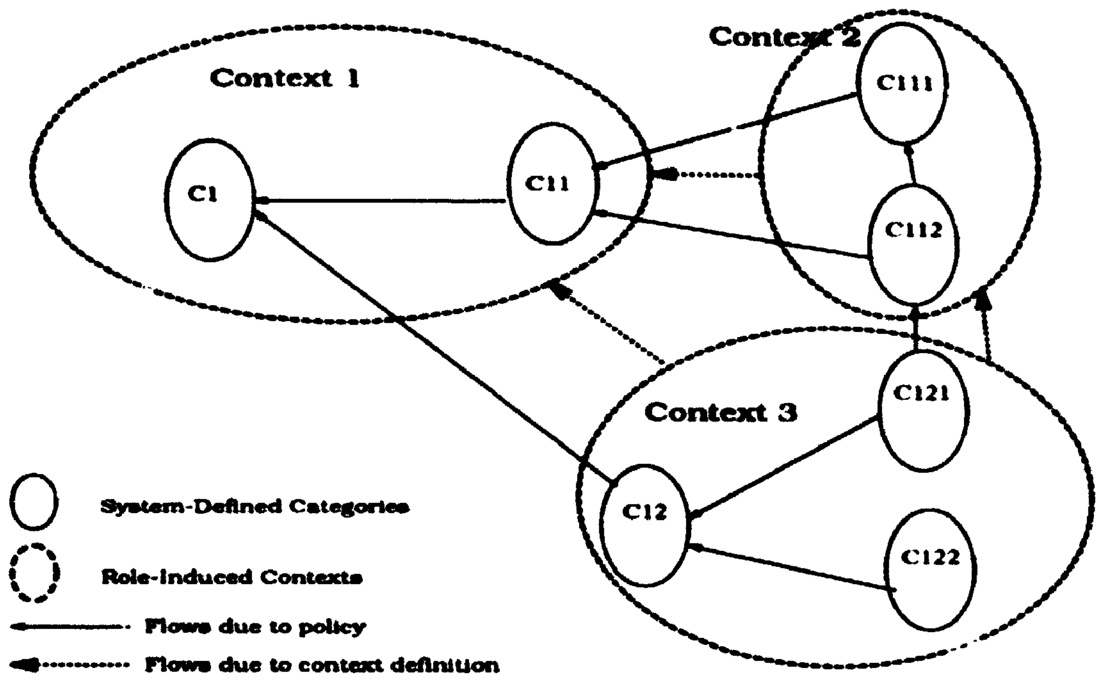

Figure 5.5: Policy & Context Information Flow

more that one system-defined category. It is possible that several categories belong to one context. Moreover, several contexts can belong to one category (see figure 5.6).

Different relationships can exist between contexts induced by role definition. They may or may not overlap. A context can be a proper subset of another context or indeed, a context can be equivalent to another one when they are associated with roles with equivalent privilege sets.

5.5.4.3 Roles, Contexts & Information Flow

Role executions may cause changes across system-defined information categories, just as they can cause such flows across role-induced contexts. In other words, execution of one role can cause change in information in different system specified categories. Moreover, such changes could involve information accessible via another role. There is bound to be information flow across system defined categories as well as across role-induced contexts. By examining these overlaps and the nature of the operations involved, we can carry out information flow analysis to determine the impact of these information flows on system security. This is important in order to determine whether a given role scheme preserves system security.

We start with the basic observation from the confinement property (see property 5.2) which implies that information flows within a category (definition 5.4). In

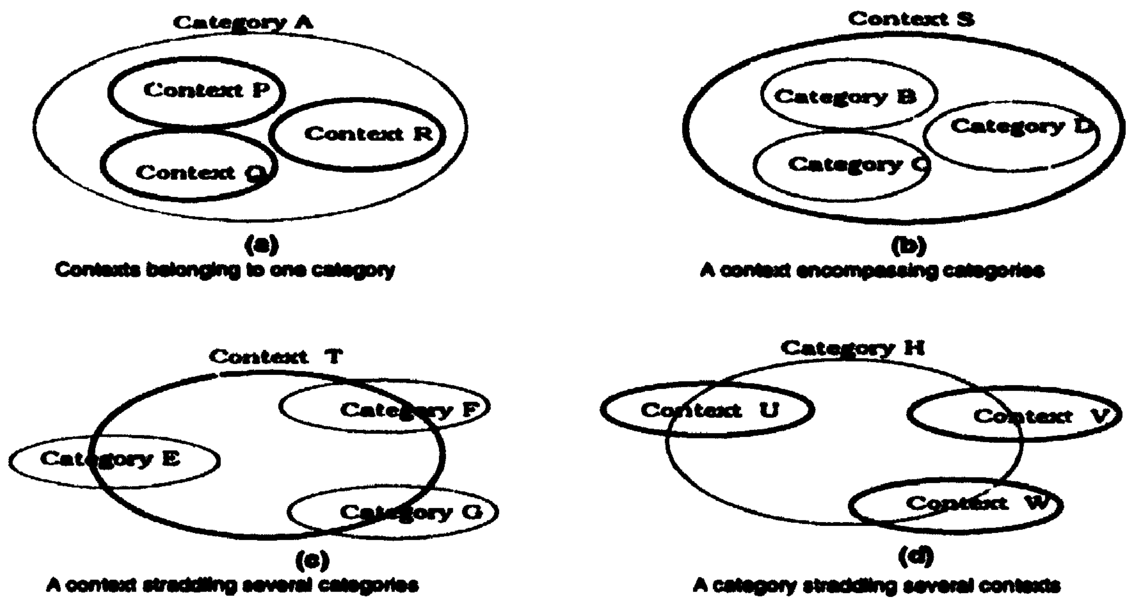

Figure 5.6: Category-Context Relationships

other words for two objects in some category $c \in \mathcal{C}$, information can flow from one object to the other, i.e. $\forall o_i, o_j \in c$, flows of the form $o_i \longleftrightarrow o_j$ are legal.

To analyze information flows from one category to another, one must analyze the operations involved. In particular we have the the basic information flow axiom (see axiom 5.1) which holds that there is information flow when there are updates. Two kinds of operations can be found in systems: read operations which do not alter the state of the associated objects and update operations which do alter/update/write states of the associated objects. An operation that does reads and updates can be seen as causing information to flow from the inputs that are read to the outputs that are updated.

Update operations within a role-induced context can be seen as defining the context's update scope. It is via these operations that information associated with a context is altered. It is via such operations that update effects of the operations can be felt in other categories. Hence it is via the same operations that information can flow from one context to another context. In a similar manner, we can specify the read scopes of roles. A role's read scope facilitates access to information via the role without causing any side effects. We designate the read and update scopes by *r_scope* and *u_scope*, respectively. Hence for some role $r \in \mathcal{R}$, r_scope(r) and u_scope(r) refer to the read and update scopes of r, respectively.

From the basic information flow axiom we can make some general statements about information flows using the read and update scopes for a given role. Clearly, an operation that does both read and update would cause information to flow from the role's read scope to its write scope. Moreover, for an operation that does multiple updates, information flows from within the update. We have the following generalizations for a given role $r \in \mathcal{R}$ with r_scope(r) and u_scope(r):

1. $r\_scope(r) \longrightarrow u\_scope(r)$ for some $r \in \mathcal{R}$. This implies there is information flow within a role, i.e. $r \longrightarrow r$ which is in agreement with the confinement problem.

2. $u\_scope(r) \longrightarrow u\_scope(r)$ for some $r \in \mathcal{R}$. This kind of flow, like the one before is in agreement with the confinement problem.

3. $r\_scope(r_i) \not\longrightarrow r\_scope(r_j)$ for $r_i, r_j \in \mathcal{R}$, i.e. there is no information flow across read scopes of two different roles.

4. Where there is overlapping of scopes of different roles we can expect information to flow between the two associated contexts. Consider two roles r_i and r_j with the following scopes: r_scope(r_i)={x,y,z}, u_scope(r_i)={p,q} and r_scope(r_j)={a,b,c}, u_scope(r_j)={d,e,f} for role r_i and r_j, respectively. From item 1 above we have: {x,y,z}$\longrightarrow${p,q} and {a,b,c}$\longrightarrow$ {d,e,f} which are information flows due to r_i and r_j, respectively.

   Information flows from r_i to r_j, i.e. $r_i \longrightarrow r_j$[4] if and only if the updates due to r_i are accessible via r_j. In other words, in the above example, either p or q or both belong to r_j's scope. Hence if we have $r_i \longrightarrow r_j$ then either u_scope(r_i) $\cap$ r_scope(r_j)$\neq \emptyset$ or u_scope(r_i) $\cap$ u_scope(r_j)$\neq \emptyset$.

   Given two roles, information can flow from one into the other and vice versa. For example, with roles r_i and r_j, we have a bidirectional information flow if information flows from r_i to r_j and vice versa, i.e. we have both $r_i \longrightarrow r_j$ and $r_j \longrightarrow r_i$. Bidirectional information flow is designated as $r_i \longleftrightarrow r_j$. We can also have multidirectional flows where we have more than one role.

   A unidirectional information flow can be either $r_i \longrightarrow r_j$ or $r_j \longrightarrow r_i$, but not both. Suppose it is required that we have $r_i \longrightarrow r_j$, it follows that u_scope(r_i) $\cap$

[4]More precisely, information flows from the context of r_i to that of r_j, denoted $context(r_i) \longrightarrow context(r_j)$. The notation $r_i \longrightarrow r_j$ is used here as a shorthand notation for convenience.

r_scope(r_j)$\neq \emptyset$ or u_scope(r_i) $\cap$ u_scope(r_j)$\neq \emptyset$ and u_scope(r_j) $\cap$ r_scope(r_i)$= \emptyset$ or u_scope(r_j) $\cap$ u_scope(r_i)$= \emptyset$.

A subsumed flow is where information flows from a context into exactly one context. Formally:

Definition 5.10 Subsumed Information Flow: *A subsumed information flow is one in which all flows from one context flows into* exactly one *context.* □

Suppose such a flow is of the form $r_i \longrightarrow r_j$. Then the conditions for such a flow are: either u_scope(r_i) $\subseteq$ r_scope(r_j) or u_scope(r_i) $\subseteq$ u_scope(r_j).

5.5.5 Roles & Information Flow in Role-Based Systems

In the foregoing sections we defined the meaning of information flow across contexts associated with given roles. In this section we use the results to discuss the management of information flows using role-based protection schemes. Consequently, we present an outline for designing role-based security schemes using information flow concepts.

In an information flow role-based protection system, we must:

1. specify what categories of information are to be handled by the system. For each information bearing entity/object/resource, we specify into which of the categories it belongs. We say such categories $\mathcal{C}(\mathcal{P})$ are created based on the security policy $\mathcal{P}$. Thus we have that $\forall o \in \mathcal{O}, \exists c(p) \in \mathcal{C}(\mathcal{P}) | o \in c(p)$.

2. specify the legal information flows allowed in the system. These flows and those inferred using the axioms 5.1 and 5.2 from their specification we refer to as $\mathcal{F}(\mathcal{P})$ since they result from the security policy $\mathcal{P}$.

   This can be seen as specifying constraints on information flow among objects/resources/etc. For instance we could explicitly say that there is necessary flow of information between a voucher object to a cheque object. This lays down the information flow policy and allowable information flows.

3. specify intended roles. The intended roles yield flows which can be termed the *explicit* flows due to the implementation. Roles are specified by the objects they are associated with and some operations on the objects.

   In general, we end up with series of information flows among defined categories which result due to both role definition and explicit specification associated with roles and their contexts. Call these contexts and flows due to contexts, $\mathcal{CO}'$ and $\mathcal{F}(\mathcal{CO}')$, respectively.

Once defined, roles determine information contexts in the system. As seen earlier, there is necessarily a flow of information within an information context if there are any updates involved. It is important that neither the implicit nor the explicit information flows violate the constraints specified in item 2 above.

$\mathcal{F}(\mathcal{CO}')$ must now be mapped to the system-defined categories to determine what flow they cause across these categories. To do so involves the superimposition of categories on the objects associated with $\mathcal{F}(\mathcal{CO}')$ (see figure 5.3). The result of this process yields flows $\mathcal{F}(\mathcal{CO})$ pertaining to categories $\mathcal{CO}$.

4. Reconcile flows in items 2 and 3. There is a contradiction of the security policy if there are flows in 3 without corresponding flows in 2. We say the two flow schemes in 2 and 3 are ***policy consistent*** if and only if none of them is contradicted. Definition 5.7 requires that $\mathcal{F}(\mathcal{CO}) \subseteq \mathcal{F}(\mathcal{P})$ for there to be consistency. Since $\mathcal{F}(\mathcal{CO})$ is the set of flows due to a realization, it is equivalent to $\mathcal{F}(\mathcal{I})$ of constraint 5.1.

   For a system to be secure, the implementation must be consistent with the policy. (Note that completeness need not imply consistency, unless there ; equivalence (definition 5.6).) As seen elsewhere consistency is a sufficient condition for security. It need not be complete.

An interesting investigation of this is to consider a set of roles $\mathcal{R}$, a set of authorizations specifying the privileges possible via the roles $\mathcal{PV}$ (with resultant information flows $\mathcal{F}(\mathcal{CO})$) and a set of legal information flow specifications $\mathcal{F}(\mathcal{P})$ (where $fl \in \mathcal{F}(\mathcal{P})$ is of the form $r_i \rightarrow r_j$ with the arrow indicating the direction of information flow) between roles and ask the question whether there will be a security violation.

Problem 5.4 *Given an arbitrary collection of privileges $\mathcal{PV}$, an arbitrary collection of roles $\mathcal{R}$ (or role definition scheme) and an arbitrary collection of system-defined categories $\mathcal{CO}$ with flows $\mathcal{F}(\mathcal{P})$ among them (this represents the system flow policy), is there an illegal information flow? In other words, do the roles cause information flows not permitted by the system security policy? Or is there a flow fl that is not in $\mathcal{F}(\mathcal{P})$.* □

To solve this problem we use graphical analysis to determine whether information flows resulting from role definition are *consistent* (definition 5.7) with $\mathcal{F}(\mathcal{P})$. Hence we construct a graph $G_1 = (V_1, E_1)$ to represent the system-specified information flows. The nodes V_1 are the system-defined categories while an edge $\langle co_i, co_j \rangle \in E_1$ whenever there exists a flow from category co_i to co_j in $\mathcal{F}(\mathcal{P})$. The graph G_1 represents the system-defined information flows and hence the system flow policy.

Next we construct another graph $G_2 = (V_2, E_2)$ where the nodes V_2 are the role contexts and the edges E_2 are the information flows between contexts resulting from role definition (see description below). Hence we have an edge $\langle context(r_i), context(r_j) \rangle$ whenever there is an information flow from the context associated with role r_i to another context associated with role r_j. The graph G_2 represents the information flows due to role definition and hence an implementation.

To determine whether or not flows in G_2 are consistent with those in G_1, we must superimpose the system categories on G_2. Thus from G_2 we derive a another graph $G_2' = (V_2', E_2')$ where the nodes are system-defined categories and the edges are derived from G_2 (see description below). Hence we have an edge $\langle co_i', co_j' \rangle \in E_2'$ if and only if there is an information flow from category co_i' to category co_j' derived from the edges E_2 of G_2.

Then using matrix analysis (see page 93), this problem can be solved in $O(n^2)$ time where n is the number of system-defined categories.

Construction of Implementation Graph G_2:

G_2 is the information flow graph derived from a role definition scheme and can be constructed statically for purposes of analysis. This is possible because privileges and roles are defined *a priori* since their effects are intended to be predictable. It is therefore possible, given a role definition scheme, to map out all potential flows that can be caused by execution of role privileges.

In constructing G_2, all operations in the role definition scheme must be analyzed (see figure 5.3). With the use of the basic information flow axiom (axiom 5.1), we consider all the operations defined by the role definition scheme that cause change in their parameters. Further, we consider all secondary invocations and side effects resulting from the invocation of the operation. These side effects in general include invocation of other operations, creation of objects, etc. according to the method invocation tree (see figure 3.2) of chapter 3.

This task of analysis may not be trivial. For each role, this analysis involves

Algorithm 5.1 *Execution Invocation Analysis*

```
For each role do
    For each privilege do
        For each mode do Analyze_Operation;
    end;
    Analyze_Operation:
        Switch Arguments;
        Case Arguments of:
            parameter: do if update, include flow from source
            created object: insert correct flow
            invocation: Analyze_Operation
        end; /* Case */
```

Figure 5.7: Role Definition Information Flow Algorithm

examining each privilege and considering the information flow(s), if any, it causes. We obtain G_2 by applying this analysis to all roles in the system and augmenting the flows caused by individual roles. G_2 is the information flow graph capturing flows across roles and their contexts. The edges in G_2 are the information flows while the nodes are the role contexts.

Central to this analysis is the nature of the operations involved. This analysis must not only determine whether an operation causes information flow but also establish the direction of the flow. Given the complex nature of operations possible in our formulation, such as O-O method invocations, this analysis must be applied recursively to all side effects, directly or indirectly, arising from a some operation invocation. Therefore, for a given operation involving a particular object, we must consider the effect of the operation on (1) the object; (2) the input and output parameters; (3) the operations it invokes (if any) and their subsequent effects; (4) the objects it creates, if any, plus the categories to which they belong, etc. In the most general sense, this analysis must consider all the paths in the operation invocation tree (similar to O-O method invocation tree of chapter 3).

The algorithm 5.1 of figure 5.7 outlines the form of the analysis.

Construction of Implementation Graph G'_2:

The graph G'_2 is derived from G_2 by superimposing the system defined categories on the role contexts. This involves taking all objects involved in G_2 and grouping them into their respective system-defined categories (see figure 5.5). Having done so, we must then analyze what flows take place from one category into another. Suppose we have objects a and b in category A and objects x and y in category X. Further, suppose that the flows discerned from G_2 show that $x \longrightarrow a$ and $y \longrightarrow b$. It follows that there exists a flow $X \longrightarrow A$.

This category by category analysis yields an information flow graph G_2' with system-defined categories as nodes and flows between them as edges.

Notice that it is necessary for the graphs to be acyclic where secrecy is required. This is the case with the Bell and LaPadula model [BL75] and Denning's Lattice model [Den76]. However, in cases where secrecy is not a priority, what is important is that the legal information flows are "honoured" and the appropriate permissions executed accurately and correctly.

5.6 Roles & Mandatory Access Control

The basis of mandatory access control is that a subject's access to an object is based *solely* on the subject's and object's attributes. Neither subjects nor objects can alter their security relevant attributes. Moreover, only authorized subjects (such as a system security officer) can alter this information. Security in this sense is mandatory in the sense that meeting the criteria based on attributes is system-enforced. There, at times, arises confusion when talking of mandatory access control which some take to be analogous to multilevel security. Hence, for purpose of this work, we define mandatory access control as follows:

Definition 5.11 *Mandatory Access Control (MAC): MAC is access control where subject access to objects/information is determined by both subject and object attributes only. Access is determined by the system which specifies access information and which forms the sole basis of access.* □

As discussed in section 5.5, roles offer segmented access to database information. Information accessed via a role this defines its context. We say that the roles *partition* information into contexts. Each context of information is accessible via subject authorization to the role associated with the context. Further, for each role, there is a subset of information accessible via the role that forms its read scope (read-set) and another subset that forms its update scope (write-set) (see figure 5.8).

To enforce MAC using roles, the contexts of information accessible via each role must be regarded as *distinct categories* of information (see figures 5.4 and 5.9). Using concepts of information flow discussed in the previous section we can then specify the legal information flows between these contexts of information. This ensures that subject authorization to a role does not violate the specified legal information flows. This is important given that the "invention" of MAC was due to the need to "tame" (confine) *Trojan horse* attacks. Trojan horse attacks involve updates and subsequent read access. A Trojan horse can encode information in one context, and then transfer it to another context where another subject is able to read it. This transfer of information violates security when it is not an authorized information flow. Moreover, since every execution in a system must be authorized, the Trojan horse action violates security.

Figure 5.8b illustrates a case where we have information confinement which meets the requirements of item 4 on page 100.

Another important consideration is that of user authorization. A user authorization scheme must ensure that user access to information does not cause illegal information flows. For example, a user authorized to two or more roles can cause information flow as in the case where the user reads information via one role and updates information in another role.

Axiom 5.3 *A user-role authorization scheme is correct if it is consistent with the specified flow policy.* □

In the Bell & LaPadula Model [BL75], information is allowed to flow up in the classification levels. The *-property bars write-downs while the simple security property bars read-up. This property ensures that a Trojan horse encoding information at high security level and trying to write it down to a lower level will not succeed. The *-property governs the direction of information flow and aims at ensuring that no *Trojan horse* program can downgrade information. Hence we have a unidirectional information flow. An important emphasis of this model is *secrecy*.

Unidirectional flows are necessarily *acyclic*. Indeed, this is the case with Denning's [Den76] lattice model, where categories of information and specified legal flows determine the category into which information from one or more categories can flow. The result is a partially ordered set of categories with a flow relationship between them. Attempts to impose a flow that is illegal are rejected.

With these two models we end up with *multilevel* security which is the most common realization of mandatory access control. Subject and object security levels

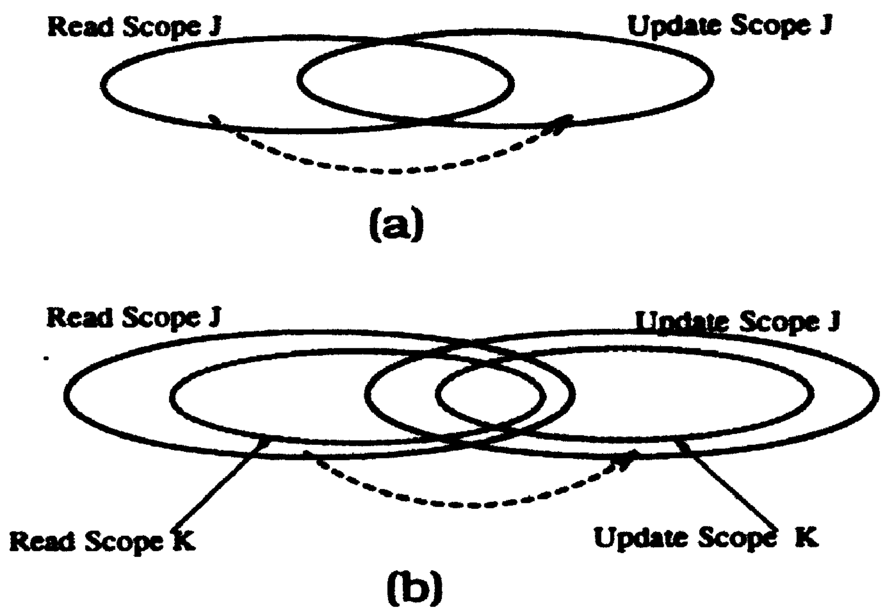

Figure 5.8: Information Flow Across Read & Update Scopes

are the relevant security attributes used to determine information access. In both models, the simple security and *-properties are used as the basis for information access. Moreover, both models depend on read and write operators.

To emulate mandatory access control in role-based protection systems, we must ensure that the system mandatory access function relies solely on the facts that a subject has ***authorization to a role***, that the role contains an associated privilege specifying the mode of access to the object, that the subject access is via this legal mode, and that the access does not violate the specified flow policy. Legal access of the object must be enforced, too, to ensure that the object is accessed in no modes other than those specified in the role as well as any other constraints specified on such access. Also, a flow policy must be observed since it is the criterion that determines security.

To ensure ***secrecy*** in our model, the information flow graphs of our role-based schemes must be acyclic. Where there are cycles, the set of all the roles/scopes in the cycle must necessarily be in the same context. Imposing the ***subsumed flow*** restriction creates a stricter information flow control. To limit the effect of Trojan horse attacks, we must formulate an equivalent of the *-property to govern our role-based protection schemes. Moreover we determine what security attributes govern information authorization.

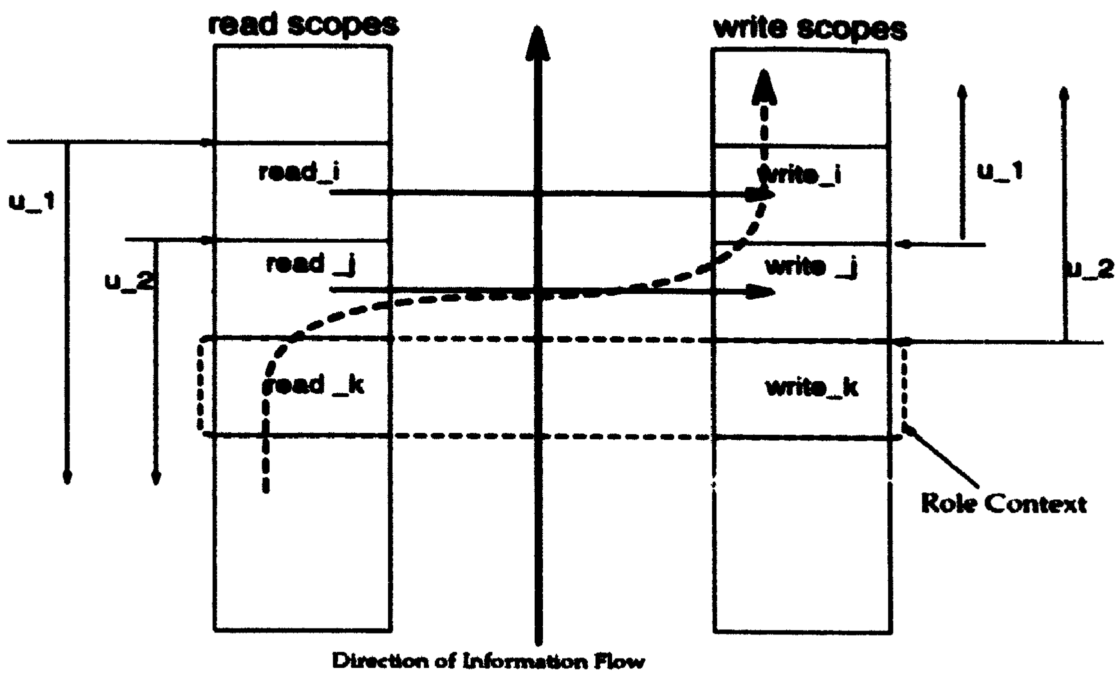

Figure 5.9: Unidirectional Information Flow in Roles

Further it must be specified what subject and object attributes would govern access of subjects to objects. Since a role facilitates subject access to objects or resources via the role, we can use this fact to specify MAC. In doing so we use two key subject/object attributes: *user authorization* to a role and *object accessibility* via a role.

Constraint 5.2 *Mandatory Authorization Constraint: A subject can only access an object via an authorized role in the mode specified in the associated privilege in the role.* □

Authorization to a role is specified in the role's access control list. Let $\mathcal{UID}$ be the set of all user identifiers, and $\mathcal{GID}$ the set of all group identifiers; $\mathcal{ID} = \mathcal{UID} \cup \mathcal{GID}$.

Definition 5.12 *Role Access Control List: A role access control list (racl) is of the form: $[id_1, \cdots, id_n]$, where $id_i \in ID$.* □

In a secure system all roles must have access control lists, i.e. $\forall r \in \mathcal{R}, \exists$ r.racl $= [\cdots, id_i, \cdots]$. We call a role with an associated access control list, *secure*.

Definition 5.13 *Secure Role: A secure role is a named collection of privileges along with its access control list. It is a triple (rname,rpset,racl), where rname is the role name, rpset is its privilege set and racl is its access control list.* □

Determining whether there is authorized access for a given user to some object in some access mode is a two stage process. First we ensure there is user-role authorization, i.e. the user's/group's identifier is in the role's access control list. Secondly, we ensure that the desired access mode to the object specified exists in the privilege set. The latter can be termed *role-privilege* authorization.

This implies a two stage process to confirm authorization: that the subject is authorized to a role and the role contains the associated privilege for access to the object. The latter is termed *object accessibility* via a role. The mode of access specified in the associated privilege is referred to as the legal mode of access via the role.

User authorization to a role means that the user can access objects accessible via the role via the specified (legal) modes of access to these objects. But user authorization alone is not sufficient to guarantee both secrecy and integrity of the information. It must be ensured that no such authorization will result in illegal information flow.

Constraint 5.2 itself is not sufficient to guarantee secrecy. Indeed, while determination of authorization is a system function, we must ensure that secrecy cannot be violated due to overlapping scopes. Hence the following constraint:

Constraint 5.3 *Read (Secrecy) Access Constraint: Given two users, u_1 and u_2, and two roles, r_i and r_j, let u_1 have access to both read scopes and u_2 have access to the read scope of r_j. Then $r\_scope(r_j)$ must be a subset of $r\_scope(r_i)$, i.e. $r\_scope(r_j) \subseteq r\_scope(r_i)$.* □

Recall that information flows from a role's read scope into its update scope, i.e. $r\_scope(r_i) \longrightarrow u\_scope(r_i)$ and $r\_scope(r_j) \longrightarrow u\_scope(r_j)$. Suppose that $r\_scope(r_j) \not\subseteq r\_scope(r_i)$. Then it means that there is information in $r\_scope(r_j)$ outside $r\_scope(r_i)$. But given that u_1 is authorized to both $r\_scope(r_i)$ and $r\_scope(r_j)$, we can have $r\_scope(r_i) \longrightarrow u\_scope(r_j)$. Hence if $r\_scope(r_j) \not\subseteq r\_scope(r_i)$ then there is information in $r\_scope(r_i)$ that is not guaranteed to flow into $u\_scope(r_i)$. In other words, u_2 has access about r_i that can be made to flow elsewhere, unless this information is a subset of r_i's read scope.

In specifying legal information flows and user authorizations, we must ensure that the read and write operations performed via different roles do not violate the specified flow policy. In other words, it should not be possible for a Trojan horse acting legally (via authorized writes) to leak information to an unauthorized context. The following two constraints are intended to guard against Trojan horse attacks:

Constraint 5.4 *Update Access Constraint: A subject cannot access one role's read scope and update another's update scope if there are no defined legal flows from the first role to the second.* □

The purpose of constraint 5.4 is to ensure that an information flow is defined in the direction of the update. This is due to the basic information flow axioms which say that information flows when there are updates.

Constraint 5.5 *Read/Update Constraint: A subject can access one role's read scope and update another's update scope if and only if the read scope of the second role contains the read scope of the first one.* □

In other words, given two roles r_i and r_j, subjects can write via r_i what other subjects in r_j can read if and only if there is defined a legal information flow (directly or indirectly) from the information context specified via $Context(r_i)$ to that specified via $Context(r_j)$.

Given that we have information of the form: $r_i \longrightarrow r_j$, the need for secrecy requires that r_j's read scope contain r_i's read scope.

From the foregoing, we conclude that MAC-like protection can be realized using role-based security if role definition and user-role authorization obey constraints 5.1, 5.2, 5.4 and 5.5. These constraints ensure an implementation with respect to a given policy, they govern user-role authorization as well as the nature of access to information via the authorized roles.

Our result is similar to that of Thomsen [Tho91]. While ours focused solely on general emulation of MAC in role-based protection by considering information flow, Thomsen's approach is with respect to well-formed transactions (WFTs) [CW87] and the read and write sets of roles and their relationships. A role's write-set (read-set) is the context of objects (information) in which can be written (read) by subjects authorized to the role. The proposed *Role-Based Security Property* states:

Property 5.1 *The Role-Based Security Property (RSP) [Tho91]: Given two roles r_1 and r_2, subjects can write via r_1 what other subjects in r_2 can read if:*

1. a subject in r_2 can read any entity that r_1 can read

2. r_1 can only use WFTs to write entities readable by r_2, or

3. r_2 can only use WFTs to read entities written by r_1. □

In the absence of WFTs (see property 5.1), only the first item is useful here. This is a formulation we define based on the concept of *information flow.*

5.7 Summary & Key Contributions

This chapter presented the concept of roles and its basis, the privilege. Consequently, we defined the terms privilege and role. We outlined the role-based protection approach, its advantages and its shortcomings.

A privilege is defined by two items: an object name and at least one of its access modes. A privilege defines what kinu of access an authorized user can have to the associated object via the associated privilege. It is a collection of objects and some of their associated access modes. A role is a collection of privileges. It determines what kind of access an authorized user can have to the associated objects via the role.

Roles can be used as the basis for system protection in which case users are authorized to specific roles. The main advantage with roles is the flexibility with which system access rights can be administered. A variation on a role's privileges varies the access rights of authorized users. As well, revocation of a user's authorization denies such a user the privileges associated with the role. Flexibility can be enhanced further should users be organized into groups and such a group authorized to a role. Hence by varying a user group's membership, we effectively alter the manner of distribution of system access rights.

If flexibility is the key advantage of role-based protection, the complexity is its price. Access rights administration and analysis is complex in role-based systems. The lack of some formal theoretical structures has hampered the ability to do this analysis. With some underlying formalism for role organization, one can design tools for access rights administration. Indeed, this work aims, in part, to iay some theoretical foundations for role-based protection schemes.

This chapter also discussed role-based protection with reference to traditional security approaches. In particular, we addressed role-based protection and information flows where, using the con ·pt of information flow, we determined constraints for ensuring security in role-based systems. We also discussed a means of realizing the equivalent of traditional mandatory access control. Hence we presented a model for multilevel security in role-based systems.

Contributions of this chapter include the formal development and formulation of the role concept. We formally defined the concept of privilege which is the underlying issue in role-based protection schemes. Consequently, we presented the formal definition of role and discussed its advantages and disadvantages in its application to security. Further, we developed an information flow analysis methodology and the security constraints for realizing security in role-base security systems. And to demonstrate the power of role-based schemes, we illustrated the realization of traditional mandatory access control using role-based protection schemes.

CHAPTER 6

ROLE ORGANIZATION

6.1 Introduction

Roles and role-based protection have been discussed in chapter 5. This chapter presents a study of role relationships based on which a framework for role organization is derived. The resulting structure forms the framework for role management and access rights administration. Our intention is to identify and define *basic* role relationships. We then use these relationships to model role organization with the ultimate aim at providing a role organization model that facilitates the ease of access rights administration in role-based protection systems. For access rights administration and role management in the proposed role organization model, associated algorithms are proposed.

The basic role relationships abstracted include: *partial, common* and *augmented* privilege sharing. A role's privilege set can be a subset of another role's privilege set. This **partial** privilege sharing is represented by the *is-junior* relationship which refers to a *junior-senior* association between two roles. Two roles can have another role that has a partial role relationship with each of them. We call this **common** privilege sharing and refer to it as *common-junior* relationship. This is the case where two (or more) roles have the same (or set of) junior role(s). The third kind of inter-role relationship is referred to as **augmented** privilege sharing where two or more roles have a common superior; each of the two roles has a partial relationship with the superior role(s). The term *common-senior* designates this type of role association. Other important concepts are those of *minimum* and *maximum* privileges. The former refers to the minimum privileges allowed for each role while the latter refers to the cumulative privileges in the system.

From these relationships we derive role structural properties which form the basis for a structure for role organization. An important property arising from these relationships is the *acyclicity* of the resultant graph structure.

Another property is the *monotonicity* of role privileges on any given path. Further, we characterize the structure by what we refer to as *coupling* based on the extent of privilege sharing among roles.

Once defined, a role graph structure requires a means for manipulating the roles and privileges in the system. To facilitate this, we explore the issues of role management and provide algorithms for the tasks. In particular, we address issues of deletion, addition and partition of roles in the role organization structure.

In section 6.2 is a review of the common role organizing structures, their similarities and differences. In particular, we focus on the common properties found in these structures. We present this information to provide the basic information for our relationship extraction. Examples of structures discussed include Ntrees [San88, San89], lattices [RWK88, RWBK91], hierarchies [TDH92], named protection domains, NPDs [Bal90] and domain definition tables, DDTs [Tho91].

Section 6.3 discusses access rights administration in role-based systems with a view to underlining the gigantic task of access rights specification and management. This forms the justification for proposing a structure for role organization. It is from this discussion and that on role organizing structures that we extract the properties that we need for role organization.

Section 6.4 discusses the basic role relationships and their utilization in role organization. We explore the concepts of partial, common and augmented privileges. We enunciate the notions of and semantics of maximum and minimum privileges. This leads to the derivation of the role graph structure presented in section 6.5. First, we discuss the structure informally and then formalize it by specifying the operator semantics that represent the basic role relationships and the properties of the role graph structure. Other interesting explorations in this section include the distribution of privileges in the model and means of deducing the privileges associated with the roles in the system.

Section 6.7 is on role administration in which we discuss role deletion, addition and partition. Corresponding algorithms are also presented. A key observation is that the model properties must be maintained whenever there is a change in structure. Moreover, we require that such changes be carried out by transactional executions.

In section 6.8 we present a comparison of our model and the other organizing structures. We explore how our model could simulate the other structures. The major advantage with our approach is that we formally characterize the role organization structure. In this section we demonstrate that it has the expressive power of other role organization structures. Section 6.9 is the summary and contribution of this chapter.

6.2 Role Organizing Structures

This section offers a brief discussion of role organizing structures with emphasis on their properties. A more detailed account is in appendix A. We start with a discussion of the similarities and differences between the different structures. As will be seen in the presentation, structures that capture both the *function* and *structure* [DM89] of a role have specific properties that will be useful in modeling role organization. Among these are two key properties: *acyclicity* of the organization structure and *increasing monotonicity* of the privilege function for roles in a given path. These will form the basis of the formulation of our own role graph structure in the next sections.

6.2.1 Organizing Structures: Similarities and Differences

Two key issues can be seen as completely specifying a role: its *function* and *structure* [DM89]. Function specifies what an authorized user can do while executing in the role. Structure defines the role's relationships with other roles in a given system. The two, however, are *not orthogonal*, given that a role's function depends, to a large extent, on its relationships with other roles. This is due to the fact that a role's functionality depends both on the privileges explicitly specified in the role and those implicitly deduced from its relationships with other roles. Hence role functionality directly depends on the role structure which characterizes role relationships. However, role structure need not depend on role functionality.

Different formal structures with known mathematical properties have been suggested for the realization of role-based protection. Structures such as lattices [RWBK91, RWK88], domain definition tables (DDT) [Tho91], Ntrees [San88, San89], hierarchies [TDH92] and privilege graphs (PGs) [Bal90] have been proposed for role organization. That we have such a variety is because of the flexibility of role-based protection to be adaptive to different policies and organizing structures. We shall only give an overview of these structure in this section. Appendix A on related work provides a more detailed account.

Ntrees [San88, San89] have properties similar to lattices and can be transformed into ordinary hierarchies via appropriate procedures. Interested readers are referred to [San88, San89, San91]. Indeed, even lattices can be transformed to equivalent tree structures by tracing every *path* from the common upper bound to the common lower bound for all nodes in the lattice. DDTs, on the other hand, are different from any of these structures, hence we shall briefly outline their use to enforce role-based protection.

Among these role organizing structures, lattices [RWK88, RWBK91], Ntrees [San88, San89], hierarchies [TDH92] and privilege graphs [Bal90] *completely* specify the roles whose organization they model. In other words, they specify both the structure and function of the roles they characterize. The role relationships specified by the structures capture the role structure. Role function, on the other hand, is also defined within the node which represents the role via the privilege set defined in the node. Consequently, we argue that these structures completely define the roles they model.

DDTs, unlike lattices, Ntrees, User Hierarchies and privilege graphs, only specify the functional aspect of a role. Indeed, a DDT structure captures both the role function as well as the authorization to the role. While it is not explicit, except for the privilege graph, how this authorization information is specified in the other structures, it is safe to assume that such information would be held in some form of access control list (see definition 5.13) for a role. Indeed, this is the approach we adopt for user authorization in our formulation later in this chapter. In fact, one can extend the access control list concept such that it incorporates a role's structural information. In doing so, the access control list entries would be either user, group or role identifiers. Where the entry is the identifier of some role r_i in the access control list of role r_j, it means that role r_i is authorized to the privileges of role r_j. This can be seen as specifying a junior-senior relationship in which r_j and r_i are the junior and senior roles, respectively. Baldwin [Bal90] advocates a similar approach which can rightly be called role-role authorization.

It is not clear how structural aspects of role definition can be specified within a DDT. Presumably such structural information can be expressed outside the DDT structure. See further discussion on DDTs in appendix A.2.3 on related work.

In a nutshell, some structures capture both role and function in their organization of system privileges. Authorization information is also important and needs to be catered for in role organization. Our work recognizes these distinctions and necessities.

6.2.2 Role Organization Structures & Their Properties

A key observation in role-based organization is the centrality of privilege sharing between roles in an effort to minimize the number of times privileges are specified in roles. Enumerating role privileges separately would be haphazard and would present a problem in tracking privilege distribution. This is due to the fact that roles can share the functionality, yet one's role's cumulative functionality would be different from the other role's. Based on functionality, roles can be used to determine the "authority" relationships among roles related via partial privilege sharing. Due to the transitivity of partial privilege sharing, roles related via partial privilege sharing realize a total order. Such a total order can be seen as a path (see definition 6.9) along which there is a *monotonically increasing* privilege relationship among the roles. Moreover, such a privilege distribution scheme implies user privilege overlap for users authorized different roles in the same path.

In Rabitti's lattice organization [RWK88, RWBK91] (figure 5.1), there is a *junior-superior* role relationship between two roles when there is an arc (or path) connecting them. The arc (or path) implies that the junior role's privileges (hence its functionality) are accessible to the superior role. Therefore the junior role's privileges form a subset of those of the senior one. Implied in this organization is some *chain of command* along any path with the senior role wielding greater *authority* than the junior one. Moreover, there is some minimum set of privileges (hence functionality) that is available to all roles in the lattice. The maximum privilege set signifies the cumulative privileges available in the organization to which the roles pertain.

Observe that the lattice structure is *acyclic* and the privileges associated with roles in a given path grow *monotonically* from the junior roles to the senior ones.

In Ting et al.'s [TDH92] user role definition hierarchy, URDH, there are similar relationships between roles as those in the lattice structure. Roles have a hierarchical ordering; any roles in a given path have a junior-senior relationship in which the junior role's privileges are available to the the senior role. Moreover the hierarchy itself is *acyclic*. Since the privileges of the totally ordered roles increase along the path, we say that the structure captures both the role privilege *increasing monotonicity* for a given path. As well, it captures structure *acyclicity* for the overall organizing structure.

Privilege graphs [Bal90] (figure 6.2) also have properties similar to hierarchies and lattices. The effective (cumulative) authorization of a user is the set of all privileges associated with roles along the path from the user's node to all the functionalities

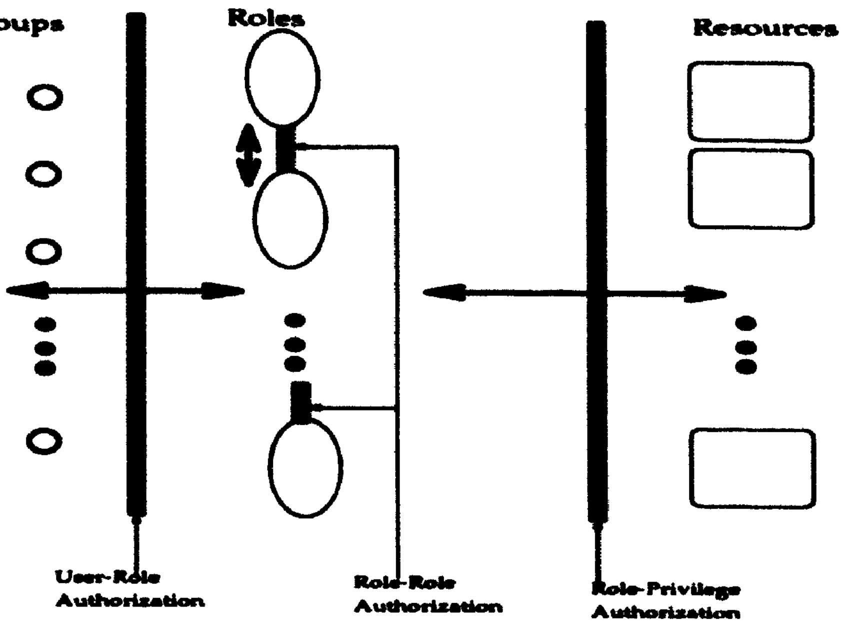

Figure 6.1: Three Kinds of Authorizations

associated with the path. The roles along such a path also have a junior-senior relationship in which the privileges (hence functionality) of the junior role are available to the senior role. Thus we have a monotonic privilege relationship for roles in any given path. Moreover the privilege graph itself is *acyclic*.

Ntrees [San88, San89] have properties similar to those of lattices. They too have both monotonicity of privileges for a given path as well as acyclicity of the structure.

DDTs, on the other hand, contain no structural information. Hence they are of no consequence to our formulation of role relationships. Role relationships in DDTs must be defined outside the DDT structure.

6.3 Roles & Access Rights Administration

Roles act as *gateways* to system information. The privilege set of a given role determines what information is available via the role. One advantage of role-based protection is that access to system information can be seen to be accomplished at two levels: via explicit authorization to a role or via inclusion of some privilege in a role. We term the former *user-role* or *group-role* authorization while the latter is termed *role-privilege* authorization (see figure 6.1).

In **user-role** authorization, a user/group is authorized access to system privileges available via the role. Such authorization must be specified in a role's access control list (see definition 5.12).

Let the set of roles in a given system be $\mathcal{R}$. We have that $\forall r \in \mathcal{R}, \exists$ r.racl $= \{\cdots, id_i, \cdots\}$. We term a role with an associated access control list *secure* (see definition 5.13). User-role authorization for a given user means that such a user (or user's identity) in the role's access control list.

Role-privilege authorization involves role configuration in which a privilege is added to the role's privilege list. **Role-role** authorization [Bal90] forms the third kind of authorization. If a role A is authorized to access a role B, it means that all of B's access rights are available via role A. In other words, B's privileges are a proper subset of the effective privileges of A. Role-role authorization can be seen as capturing role relationships; it specifies the structural component of the role.

Let $\mathcal{PV}$ be the universal set of privileges in a system. A function $\Psi : \mathcal{R} \mapsto \mathcal{PV}$ gives the privileges of a given role, i.e. given some role $r \in \mathcal{R}, \Psi(r) = \{pv_1, \cdots, pv_n\} = r.rpset$.

Example 6.1 **Suppose we have two roles: clerk and supervisor in which the supervisor role has a role authorization to the clerk role. This means that the clerk's access rights are available to the supervisor. A user authorized to the supervisor role can perform whatever a user authorized to the clerk role can do.**[1] **We can view the privilege relationships between the two roles as $\Psi(clerk) \subseteq \Psi(supervisor)$.** □

Role-role authorizations can be complex. To capture the role-relationships completely and be able to carry out analysis of implications of privilege assignment and distribution in a system can be even more complex without some formal organizational structure. Complexity of analysis of system privilege distribution is one short-coming of role-based protection [TDH92, NO93b].

Baldwin's approach to access rights administration uses privilege graphs (PG) which capture functionality, structure and authorizations. A PG (figure 6.2) is an acyclic graph with three types of nodes: functionality, role and user/group. A path from a given user node to a functionality node means that the user is authorized to execute the functionality. The access rights available to such a user are all the privileges specified in roles in any such path.

Ting et al.'s [TDH92] approach utilizes hierarchical ordering of roles in which for any given roles in a path, those lower in the hierarchy have lower functionality than

[1]Separation of duty [CW87], on the other hand, can be specified to ensure that the supervisor does not perform both roles.

those high in the hierarchy. In general, the path captures a subsetting relationship between the roles such that for a given edge directed $\langle v_i, v_j \rangle$, $\Psi(v_j) \subseteq \Psi(v_i)$. Both of these structures have what we term the *acyclicity property*.

Definition 6.1 *Acyclicity Property: A structure of role organization is said to have the acyclicity property if a graph of their relationships, defined with the roles as nodes, is acyclic and a directed edge $\langle r_i, r_j \rangle$ implies that $\Psi(r_i) \subseteq \Psi(r_j)$.* □

Without this restriction, the union of all privileges in a given loop would be available to every role in the loop.

Constraint 6.1 *Role Organization Structure Acyclicity: A role organization must preserve the acyclicity property in order to offer differentiated access to system information via role-based protection techniques.* □

The subsetting privilege relationship among roles in a given path leads to the *monotonically increasing* privilege relationship along a given path. Formally:

Definition 6.2 *Function Monotonicity: A function f over a totally ordered set $x_1 < x_2 < \cdots < x_n$ is said to be monotonically increasing over the total order if and only if $f(x_1) \subseteq f(x_2) \subseteq \cdots \subseteq f(x_n)$.* □

Constraint 6.2 *Privilege Monotonicity Constraint: For any given path in a role organizing structure, there must be a monotonically increasing privilege relationship among roles along the path.* □

In the next section we present a role graph model which captures all these properties. It specifies minimum functionality requirements and avails it to all roles in the system. Further, it provides a means of analysing shared functionality, common superset functionality and subsetting functionality in the resulting role organizing structure. The model addresses neither user-role nor role-privilege authorizations.

6.4 Modeling Role Organization

A role is a collection of privileges which facilitates the execution of some *functionality* for an authorizeu user. Roles in a system can have different kinds of relationships among themselves based on their associated functionalities and any organizational constraints. Thus it is important to develop some formal organizational framework which expresses desirable organizational properties and, in the process, captures the relationships among roles. Such a framework would facilitate the ease of analysis of

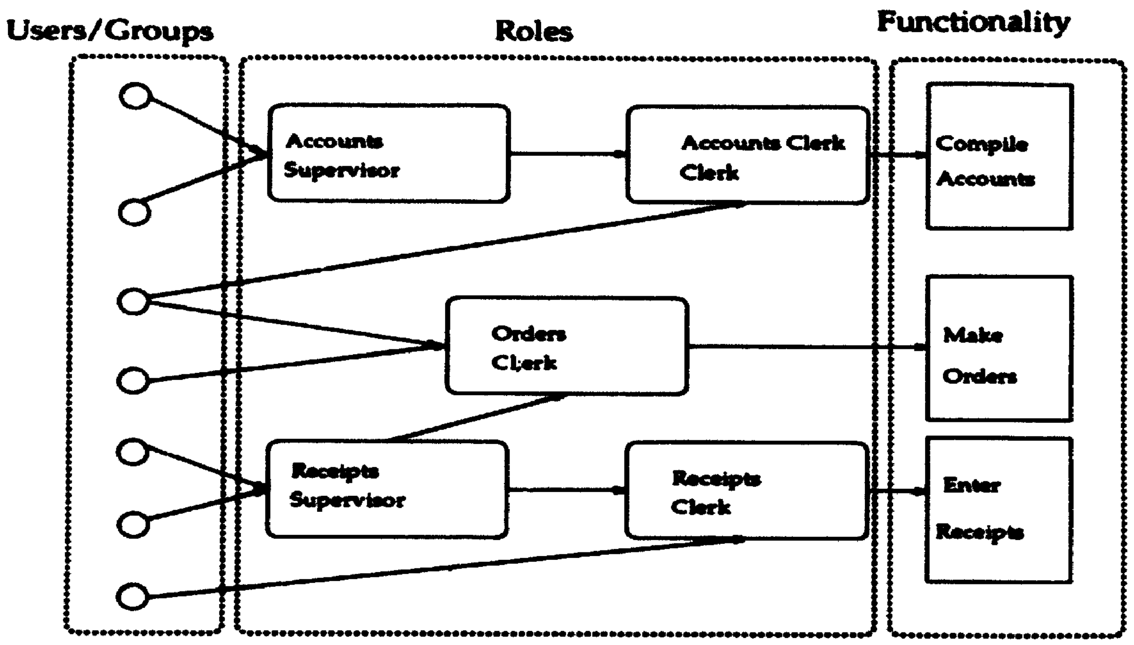

Figure 6.2: Privilege Graph

privilege distribution and sharing. It would help in the assessment of implications of privilege assignment, sharing and distribution in a system. Such implications are important, especially with respect to system security.

The motivation for this approach is based on the observation that a big drawback of role-based protection is that the analysis of privilege assignment is complex [TDH92, NO93b]. A formal framework for role organization and privilege assignment analysis would ease this task. Moreover, it can lead to formal tool development for such analyses. Role administration would be enhanced further if there are developed algorithms for role management (such as role addition and deletion) and role privilege assignment (role-privilege addition and deletion). With a formal role organization model, we can formulate algorithms which guarantee properties of the model.

In this section we discuss and model what we regard as basic role relationships which must form the basis of a role organization framework. We start with relationships between two roles, introduce the concepts of the minimum and maximum privilege sets in a role-based system and their relationship with other roles. Finally, we combine these concepts to yield a framework for role organization.

6.4.1 Role Relationships: The Basics

We shall use the following notation, which has been introduced before, throughout the rest of this thesis. We denote the finite universe of roles in a system by $\mathcal{R}$. Hence with n distinct roles in a system we have $\mathcal{R} = \{r_1, \cdots, r_n\}$. Let $r \in \mathcal{R}$ be some role in the system. Let $\mathcal{PV}$ be the finite universe of privileges in the system. $\mathcal{R}$ and $\mathcal{PV}$ are related via the function $\Psi : \mathcal{R} \mapsto \mathcal{PV}$. Given a role $r \in \mathcal{R}$, Ψ enumerates its associated privileges, i.e. $\Psi(r) = \{pv_1, \cdots, pv_m\} = r.rpset$.

6.4.1.1 Two-Role Relationships

This section address relationships between two roles. We identify three kinds: junior-senior, common "junior" and common "senior". Junior-senior relationship, referred to as *is-junior* relationship in this thesis and expressed as *junior→senior*, expresses the fact that a senior role's privileges include those of the junior one. The common junior, referred to as *common-junior*, relationship, denoted by "$\odot$", expresses the relationship between two roles whose result is another role, junior to both of them. The common senior relationship, referred to as *common-senior*, denoted by "$\oplus$", on the other hand, results in a role which is senior to both given roles. Figure 6.3 shows these possibilities with the Venn diagrams showing the associated privileges.

In all these cases, the underlying observation is that there is privilege sharing and hence functionality sharing between two roles. To analyze the manner of interaction between such roles would involve finding out their nature of interaction via role sharing. The three relationships form the following three kinds of privilege sharing that can be found in organizations.

1. **Partial Privileges**

   In partial privilege sharing, we have privileges defined in one role being a *complete* subset of privileges in another role. This implies shared functionality via the shared privileges. For instance, the **clerk** and **supervisor** roles in example 6.1 share the functionality associated with the **clerk** role, i.e. a user authorized to the **supervisor** role can execute the functionalities associated with both roles (figure 6.3a).

   We model such *direct* functionality and privilege sharing using the **is-junior** relationship denoted by "$\rightarrow$" In our example we have **clerk→supervisor**. In general, given two roles $r_i, r_j \in \mathcal{R}$ with $r_i \rightarrow r_j$, we have the following interpretation:

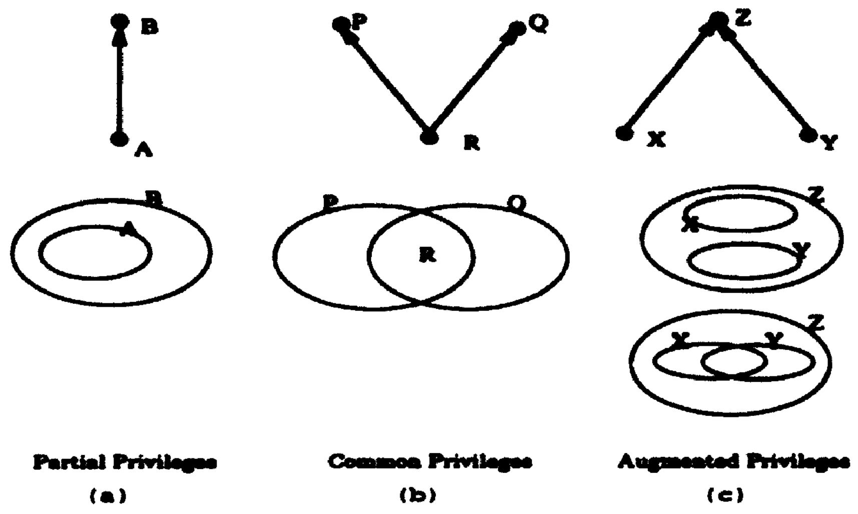

Figure 6.3: Three Kinds of Two-Role Relationships

r_i and r_j are "junior" (subservient) and "senior" (superior) roles, respectively. Moreover, r_i's privileges and functionality are available to r_j. Hence $\Psi(r_i) \subseteq \Psi(r_j)$. We say r_i's privileges are **indirectly** *available to r_j.*

Formally, the *is-junior* relationship is defined as:

Definition 6.3 *is-junior relationship ($\rightarrow$): An* is-junior *relationship exists between two roles r_i and r_j, denoted $r_i \rightarrow r_j$, if and only if $\Psi(r_i) \subseteq \Psi(r_j)$.* □

The *is-junior* relationship can be seen as a role-role authorization in which the superior role is authorized to the privileges of the junior role. Suppose we have two roles A and B with privileges $\Psi(A)=\{1,2,3\}$ and $\Psi(B)=\{1,2,3,4,5,6\}$. We have $\Psi(A) \subset \Psi(B)$. We say A$\rightarrow$B. There exists an *is-junior* relationship between A and B.

If we consider relative authority as a measure of the privileges associated with a role, then the "$\rightarrow$" relationship can be seen as specifying which of the two roles has a higher authority than the other. In our case the junior role exercises less authority than the superior one. Moreover, the "$\rightarrow$" relationship can be seen as specifying the flow of authority. Further, for this authority to be meaningful, this relationship must not violate the acyclicity property (see constraint 6.1).

2. **Common Privileges**

Another form of relationship between two roles is where there is privilege sharing in which roles have a non-empty intersection of their privilege sets but with neither of the sets being a subset nor a superset of the other. Such a relationship can be used to indicate an overlap of responsibility (figure 6.3b).

If there exists a role defined whose privilege set is some or all of this intersection, then we say such a role is a **common-junior** to the other two roles. We denote the *common-junior* relationship with "$\odot$". Suppose we have roles A, B and C related as $A \odot B = C$. Suppose the privilege sets associated with A and B are $\Psi(A)$=$\{1,2,3,4\}$ and $\Psi(B)$ =$\{3,4,5,6,7\}$, respectively. $\Psi(C)$ must be a common subset of both $\Psi(A)$ and $\Psi(B)$, i.e. $\Psi(C) \subseteq (\Psi(A) \cap \Psi(B)) = \{3,4\}$.

In general, given three roles $r_i, r_j, r_k \in \mathcal{R}$ with $r_i \odot r_j = r_k$, we have the following interpretation:

> *bot' r_i and r_j are senior (superior) roles to r_k. Moreover, r_k's privilege and functionality are indirectly available to both r_i and r_j. Hence $\Psi(r_k) \subseteq \Psi(r_i)$ and $\Psi(r_k) \subseteq \Psi(r_j)$.*

Definition 6.4 <u>*common-junior relationship* ($\odot$)</u>: *Given roles r_i, r_j and r_k, r_k is a* common-junior *of r_i and r_j denoted $r_k = r_i \odot r_j$ if and only if $\Psi(r_k) \subseteq (\Psi(r_i) \cap \Psi(r_j))$.* □

In general if we have $r_k \rightarrow r_i$ and $r_k \rightarrow r_j$ then we have $r_i \odot r_j = r_k$.

3. **Privilege Augmentation**

Another important consideration is privilege augmentation. In analysing privilege distribution it may be necessary to find a role that embodies the functionality and privileges of two given roles. Such a role would contain the privilege sets of the two given roles. Clearly, such a role's privileges will be a superset of both given roles (figure 6.3c).

The relationship in such a case is termed **common-senior** and denoted by "$\oplus$". Suppose we have roles X, Y and Z related as $X \oplus Y = Z$. Suppose we also have $\Psi(X)$=$\{1,2,3,4\}$ and $\Psi(Y)$=$\{6,7,8,9\}$. For Z's privileges to be a common superset of those of X and Y, we must have $(\Psi(X) \cup \Psi(Y)) \subseteq \Psi(Z)$, i.e. $\{1,2,3,4,6,7,8,9\} \subseteq \Psi(Z)$.

Given three roles $r_i, r_j, r_k \in \mathcal{R}$ with $r_i \oplus r_j = r_k$, we have the following interpretation:

> *both r_i and r_j are junior (subservient) roles to r_k. Moreover, both r_i's and r_j's privileges and functionalities are indirectly available to r_k. Hence $\Psi(r_i) \subseteq \Psi(r_k)$ and $\Psi(r_j) \subseteq \Psi(r_k)$.*

Definition 6.5 <u>common-senior relationship ($\oplus$)</u>: *Given roles r_i, r_j and r_k, r_k is a common-senior of r_i and r_j denoted $r_k = r_i \oplus r_j$ if and only if $(\Psi(r_i) \cup \Psi(r_j)) \subseteq \Psi(r_k)$.* □

In general, if we have $r_i \rightarrow r_k$ and $r_j \rightarrow r_k$ then $r_i \oplus r_j = r_k$.

6.4.1.2 Beyond Two-Role Relationships

The relationships introduced in the previous section can be extended to cater for more than two roles. This section outlines how the extensions can be realized.

1. **Partial Privilege Sharing**

   From the previous section observe that we can have *is-junior* relationships of the form $r_i \rightarrow r_j$ and $r_j \rightarrow r_k$. From the definition of the *is-junior* relationship, if $(r_i \rightarrow r_j) \wedge (r_j \rightarrow r_k)$ then r_i and r_k must be related thus: $r_i \rightarrow r_k$ since $(\Psi(r_i) \subseteq \Psi(r_j)) \wedge (\Psi(r_j) \subseteq \Psi(r_k)) \Rightarrow (\Psi(r_i) \subseteq \Psi(r_k))$. This then captures the *transitive* property of the *is-junior* relationship. In general, if we have a role relationship of the form: $r_i \rightarrow r_{i+1} \rightarrow \cdots \rightarrow r_{i+n}, n \geq 0$ it follows that $\Psi(r_i) \subseteq \Psi(r_{i+1}) \subseteq \cdots \subseteq \Psi(r_{i+n})$. This captures the monotonic property of the privilege function for roles related via the *is-junior* relationship.

   Property 6.1 *The privilege function Ψ increases <u>monotonically</u> with respect to the* is-junior *($\rightarrow$) relationship.* □

   We denote $r_i \rightarrow r_{i+1} \rightarrow \cdots \rightarrow r_{i+n}$ by $r_i \rightarrow^* $ for $n \geq 0$ and $r_i \rightarrow^+$ for $n > 0$. This leads to the formal definition of a path:

   Definition 6.6 <u>Role Path</u>: *A role path, p, between two roles r_i and r_x is of the form $r_i \rightarrow^* r_x$. A trivial path exists between a role and itself.* □

   Other properties of the *is-junior* relationship include *reflexivity* and *antisymmetry*. Given roles r_i and r_j, we have $r_i \rightarrow r_i$ (reflexivity) since $\Psi(r_i) \subseteq \Psi(r_i)$.

As well. if $((r_i \rightarrow r_j) \wedge (r_j \rightarrow r_i)) \Rightarrow r_i = r_j$. This follows from the observation that $(r_i \rightarrow r_j) \Rightarrow \Psi(r_i) \subseteq \Psi(r_j)$ and $(r_j \rightarrow r_i) \Rightarrow \Psi(r_j) \subseteq \Psi(r_i)$. With $\Psi(r_i) \subseteq \Psi(r_j)$ and $\Psi(r_j) \subseteq \Psi(r_i)$ and by the acyclicity property (property 6.1), it follows that $\Psi(r_i) = \Psi(r_j)$ which implies $r_i = r_j$. This is the basis for the following property:

Property 6.2 Role Privilege Set Uniqueness: *In a given's system, a role's privilege set must be unique.* □

2. **Common Privileges**

   From the *common-junior* ($\odot$) relationship in the previous section, observe that the common subset of two roles need not be an immediate junior role of both roles in question. This leads to the following lemma which expresses the relationship between the *is-junior* and the *common-junior* operators, $\rightarrow$ and $\odot$, respectively:

   Lemma 6.1 *Given that $r_k \in (r_i \odot r_j)$, then $r_k \rightarrow^+ r_i$ and $r_k \rightarrow^+ r_j$.* □
   The *common-junior* operator ($\odot$) is commutative, associative and reflexive, i.e. $r_i \odot r_j = r_j \odot r_i$, $r_i \odot (r_j \odot r_k) = (r_i \odot r_j) \odot r_k$ and $r_i \odot r_i = r_i$.

3. **Privilege Augmentation**

   As with the *common-junior* relationship, the *common-senior* relationship need not involve immediate superiors of the role under consideration. This leads to the following lemma which captures the relationship between the two operators $\rightarrow$ and $\oplus$:

   Lemma 6.2 *Given that $r_k \in (r_i \oplus r_j)$, then $r_i \rightarrow^+ r_k$ and $r_j \rightarrow^+ r_k$.* □

   The *common-senior* operator ($\oplus$) is commutative, associative and reflexive, i.e. $r_i \oplus r_j = r_j \oplus r_i$, $r_i \oplus (r_j \oplus r_k) = (r_i \oplus r_j) \oplus r_k$ and $r_i \oplus r_i = r_i$.

Notice that $\odot$ and $\oplus$, for any two or more roles, need not result in a unique role. In the general case, the result is a set of roles corresponding to common juniors and common seniors, respectively. This is one distinguishing feature between our model and a lattice organization: the greatest lower and least upper bounds are not unique.

6.4.2 The Concepts of Minimum Privilege Sets

It is possible that an organization provides a minimum set of privileges available to every user. Such a basic privilege set, for instance, can be things like the ability/permission to log onto a computer system, the privilege to get into certain organization premises, etc. In general, this minimum privilege set represents the very minimum set of privileges that any valid user/employee can be authorized to.

Since users are authorized to specific roles, it is possible to organize such a basic set of privileges into a role such that they are available via explicit authorization or via role relationships with other roles. We denote the role with the basic privilege set **MinRole**. In general, depending on a particular organization, **MinRole** can be empty.

$$\Psi(\text{MinRole}) = \begin{cases} \text{Min. mandatory privilege set} & \text{if defined} \\ \emptyset & \text{Otherwise} \end{cases}$$

A role $r \in \mathcal{R}$ with a direct relationship with **MinRole** will be expressed as: MinRole $\rightarrow r$ and, in general, $\forall r \in \mathcal{R}$, MinRole $\rightarrow^{+} r$ holds.

Property 6.3 <u>*Minimum Privilege Property:*</u> **MinRole** *is always defined.* □

6.4.3 The Concept of Maximum Privilege Set

As with **MinRole**, we envisage **MaxRole**, some system "chief executive" role, which embodies the collection of all privileges in a given system. Theoretically, a user authorized to **MaxRole** can execute any functionality using the associated privileges in whatever role they are specified. Unlike Ψ(MinRole) which can be empty, Ψ(MaxRole) can never be empty if the system is intended to accomplish anything at all.

$$\Psi(\text{MaxRole}) = \bigcup_{r \in \mathcal{R}} \Psi(r)$$

Property 6.4 <u>*Maximum Privilege Property:*</u> **MaxRole** *is always defined.* □

6.4.4 Combining the Concepts

The *is-junior*, *common-junior* and *common-senior* relationships presented in the foregoing sections, in our view, capture all manner of relationships that can be used to associate two or more roles when there is need for analysis of their interaction. The **MinRole** and **MaxRole** roles express the concepts of minimum mandatory and

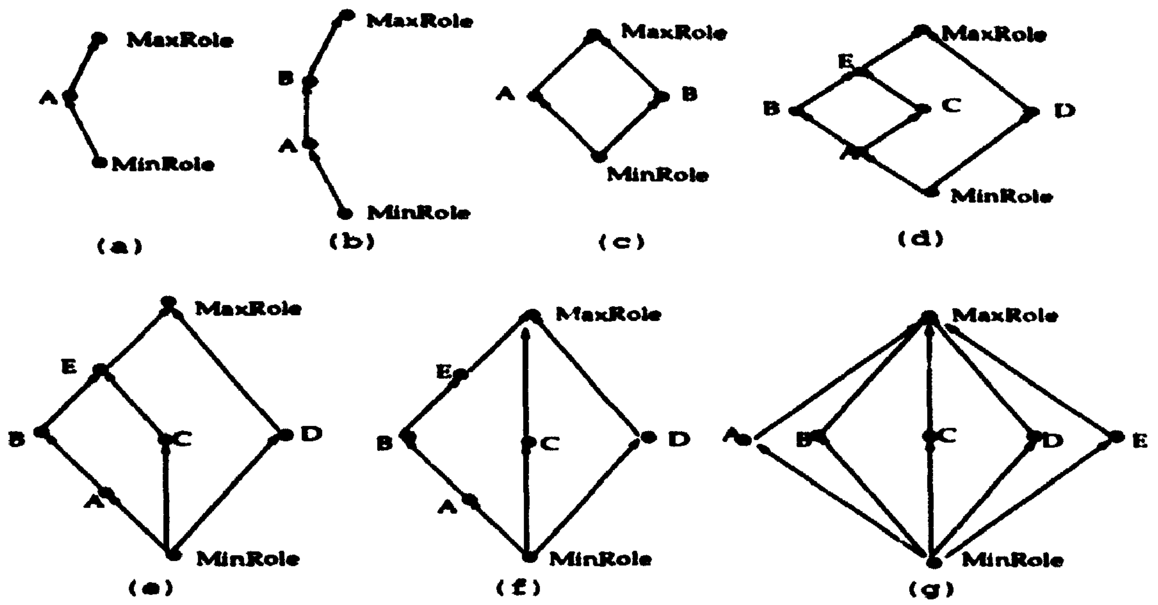

Figure 6.4: Different Forms of Role Organization

maximum privilege sets, respectively, in a system. Combining these together yields representations such as those in figure 6.4.

For the purposes of security and the need for dispersion of powers, MaxRole may not be authorized to any one individual in an organization. In an ideal situation, MaxRole conceptually corresponds with the role of a Chief Executive in an organization. It is unlikely that an administrative or a security policy would advocate such singular exercise of powers. Moreover, there is a very realistic risk that allowing exercise of privileges of MaxRole can compromise the system. However, such problems need not arise if we make the exception that no single user can exercise the privileges of MaxRole. This will make MaxRole a non-executable role. Other policies may choose a collective execution of the role, e.g. by a number of votes of authorized users. Whatever the case, authorization to MaxRole with be a matter of a specific security policy. MaxRole, in our modeling, is useful for purposes of completeness. It ensures that every two roles in the system have a common-senior just as MinRole ensures that every two roles have a common-junior.

These basic relationships, on further examination, describe a role organization in which the $\rightarrow$, $\odot$ and $\oplus$ can determine partial role ordering, greatest lower bound and least upper bound, respectively. Although our framework appears like a lattice on first sight, it is not one. However, it has properties that closely resemble those of a lattice. Consequently, we shall refer to the structure as a role graph, a kind of *pseudo lattice*. We develop these concepts further in the next section which leads to a model for role organization. Along with the model we formulate algorithms for role management including algorithms for role addition, role deletion and role partition.

6.5 A Role Graph Model for Role Organization

In this section we formally propose a model for defining the structure and function of roles in a system. The fact that it very closely resembles a lattice structure, implies that mathematical properties of lattices can be exploited in our formulation. We also notice that our *pseudo* lattice can be tailored to meet varying modes of the structure of organizations found in the real world. For instance, it is known that any employee in an organization has some basic set of rights, obligations and is accountable to some other (probably senior) employee. This concept of basic rights, obligations and accountability is captured in our role model as MinRole. Accountability is captured given that each role, except MaxRole has at least one superior role to it.

In an organization, it is common to have shared functionalities between individuals performing different tasks. Our role lattice model is able to capture this fact via the greatest upper bound role. As well, where there is a common superior role, we have the least upper bound role. As well, we easily capture cases where roles have no shared functions: role independence.

An interesting outcome of our formulation, which is also a mathematical fact of the pseudo lattice structure, is that one can define a role that captures the cumulative authority in the organization. In the real world such authority would be bestowed in either the chief executive or the board of directors.

The fact that we can express all these attributes of organization within our pseudo lattice model, makes the structure attractive. Our task here is to extract those properties of real life organizations that can be abstracted and modeled in computer systems for purposes of system protection. The properties in question, such as the way the authority flows, the functionality of various roles in the organization and the relationships of the roles with other roles can readily be expressed within the role graph model.

6.5.1 The Model: Informally

We propose a role organization modeling structure that facilitates role organization and administration of roles, which we discuss here informally and present formally in section 6.5.2. It is clear that the organization modeling approach from section 6.4 points to an acyclic graph organization. Hence our modeling will use graph theory and the properties enunciated earlier to facilitate role organization. We believe (as we shall demonstrate towards the end of this chapter) that our role graph model is at least as expressive as (if not more expressive than) other organizational structures such as hierarchies, trees, etc. We believe that we can simulate properties of these other structures using our model. Moreover, it presents no more difficult role management tasks compared to these structures. Since our model is derived from basic principles of role organization, we believe that it can approximate corporate structures in the real word. With appropriate constraints on relationships between roles, one can simulate these relationships to reflect corporate organization.

From definition 5.3, a role groups privileges that can be executed by users authorized to that role. To minimize the task of enumerating the privileges of each role, we organize them using the concepts introduced in section 6.4 which incorporates acyclicity of the role graph structure and the monotonicity of role privileges for any path. Such a structure, along with rules for role ordering and determining the privileges associated with a role, facilitate a simple, yet elegant, organization of roles to reflect the *authority*[2] attached to each role.

Role ordering and role inter-relationships, in turn, offer a means of distributing privileges among the roles. The idea is that we explicitly assign a privilege at the lowest point in the structure where it is desirable. Since our formulation specifies that high order roles can execute the privileges of the lower order ones with a connecting path (see definition 6.6), we can make the least number of explicit privilege assignments that facilitate the desired distribution.

From the ordering, we define *authority paths* that are linear (total) orders of roles according to non-decreasing authority, connected by the *is-junior* relationship ($\rightarrow$). In essence, the ordering asserts the fact that higher authority roles have access to more privileges than lower ordered ones in any given path. This is due to the fact that higher authority roles can execute the functionality of lower authority ones with a connecting path.

[2]Our use of this term will become clear as we advance.

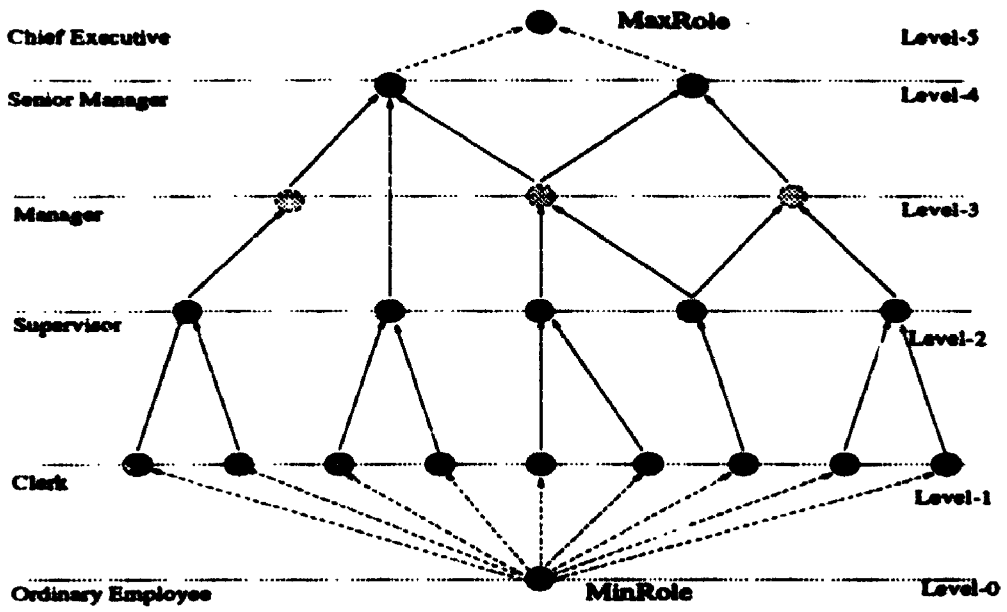

Figure 6.5: An Example of Role Organization

The effective privileges associated with a role result from those privileges *directly* associated with the role and those *indirectly* associated with it. The former are those privileges directly specified in the role while the latter are those privileges specified in lower order roles with a connecting path to the role.

6.5.2 The Role Organization Model: Formally

This section presents the formal organization of roles into a role organization structure, as shown in figure 6.5. We have a universal set of roles in a system $\mathcal{R} = \{r_1, r_2, \cdots, r_n\}$ that represents all roles available. The role graph structure, is denoted $RG = (\mathcal{R}, \rightarrow)$[3] is composed of the set of all roles on which the *is-junior* relationship imposes an ordering and in which the properties of the role relationships specified earlier hold. Hence operators $\oplus$ and $\odot$ have meaning as defined previously. They can be used to determine the relationships between any given roles in the system.

Observe that:

1. $\langle \mathcal{R}, \rightarrow \rangle$ is a partially ordered set, i.e. $\forall r_i, r_j, r_k \in \mathcal{R}$ the relation "$\rightarrow$" is reflexive, antisymmetric and transitive. $\forall r_i, r_j, r_k \in \mathcal{R}, r_i \rightarrow r_i$ (reflexivity), $r_i \rightarrow r_j \wedge r_j \rightarrow$

[3]The *is-junior* relationship ($\rightarrow$) can be seen as specifying the edges between the roles in the system.

$r_i \Rightarrow r_i = r_j$ (antisymmetry) and $r_i \rightarrow r_j \wedge r_j \rightarrow r_k \Rightarrow r_i \rightarrow r_k$ (transitivity). Hence, $\langle \mathcal{R}, \rightarrow \rangle$ is indeed a partially ordered set.[4]

The *is-junior* relationship ($\rightarrow$) can be seen as a flow relation with the arrow indicating the direction of increasing "authority". Given two roles r_i, r_j with $r_i \rightarrow r_j$, we have $r_i < r_j$ in terms of authority. This authority is defined in terms of privileges. For a given role r, its superior list is the set of all those roles $r_s \in \mathcal{R}$ such that $r \rightarrow r_s$. $\{r_s\}$ denotes this set and the relationship of r with this set can be expressed as $r \rightarrow \{r_s\}$. A *junior* list is similarly defined.

Let $r_i \rightarrow^+ r_j = r_i \rightarrow r_{i+1} \rightarrow \cdots \rightarrow r_{i+n} \rightarrow r_j, n > 0$ and $r_i \rightarrow^* r_j = r_{i+1} \rightarrow \cdots \rightarrow r_{i+n} \rightarrow r_j, n \geq 0$. We say a path (see definition 6.6) exists between two roles r_i and r_j if $r_i \rightarrow^* r_j$. Of concern to us are paths with a length greater than zero, i.e. $r_i \rightarrow^+ r_j, i \neq j$. In general, a path is a linear order of roles of the role graph according to their authority as derived from the $\rightarrow$ operator.

2. The operator "$\odot$" gives a greatest lower bounds set (glbs) of any two roles, i.e. glbs$(r_i, r_j) = (r_i \odot r_j)$. It gives roles common to the two roles involved in the operation, i.e. for any two roles r_i and r_j, if $\{r_k\} = (r_i \odot r_j) \Rightarrow \forall r \in \{r_k\}, \exists((r \rightarrow \cdots \rightarrow r_j) \wedge (r \rightarrow \cdots \rightarrow r_j))$. Or alternatively, $\forall r \in \{r_k\}$ we have $(r \rightarrow^+ r_i) \wedge (r \rightarrow^+ r_j)$ and both $((r \rightarrow \cdots \rightarrow r_j) \wedge (r \rightarrow \cdots - \;\; r_j))$ are the shortest paths of this form. Formally:

   Definition 6.7 *<u>Greatest Lower Bound Set (glbs):</u> Where $r_j \odot r_j \neq \{MinRole\}$, then a glbs exists. $\{r_k\}$ is a glbs if and only if $\forall r \in \{r_k\}, \exists((r \rightarrow^+ r_i) \wedge (r \rightarrow^+ r_j) \wedge (\nexists r_l, (r \rightarrow^+ r_l \rightarrow^+ r_i) \wedge (r \rightarrow^+ r_l \rightarrow^+ r_j))$ with both $(r \rightarrow^+ r_i) \wedge (r \rightarrow^+ r_j)$ being the shortest paths of this form . Generally, the glbs of two or more roles is a set; it is not necessarily a unique role.* □

3. The operator $\oplus$ yields the least upper bound set (lubs) of two or more roles. This is the set of the next higher roles common to two or more roles in the model. For any two roles r_i and r_j, we have $r_i \oplus r_j = \{r_k\} \Rightarrow \forall r \in \{r_k\}, \exists(\cdots r_i \rightarrow^+ r) \wedge (\cdots r_j \rightarrow^+ r)$ where both $(r_i \rightarrow^+ r) \wedge (\cdots r_j \rightarrow^+ r)$ are the shortest paths of this form.

   Definition 6.8 *<u>Least Upper Bound Set (lubs):</u> Where $r_j \oplus r_j \neq \{MaxRole\}$, then a lubs exists. $\{r_k\}$ is a lubs of r_i and r_j if and only if $\forall r \in \{r_k\}, \exists((r_i \rightarrow^+$*

[4]Edges of the form $r_i \rightarrow r_i$ are not shown in the graphs. They are implied.

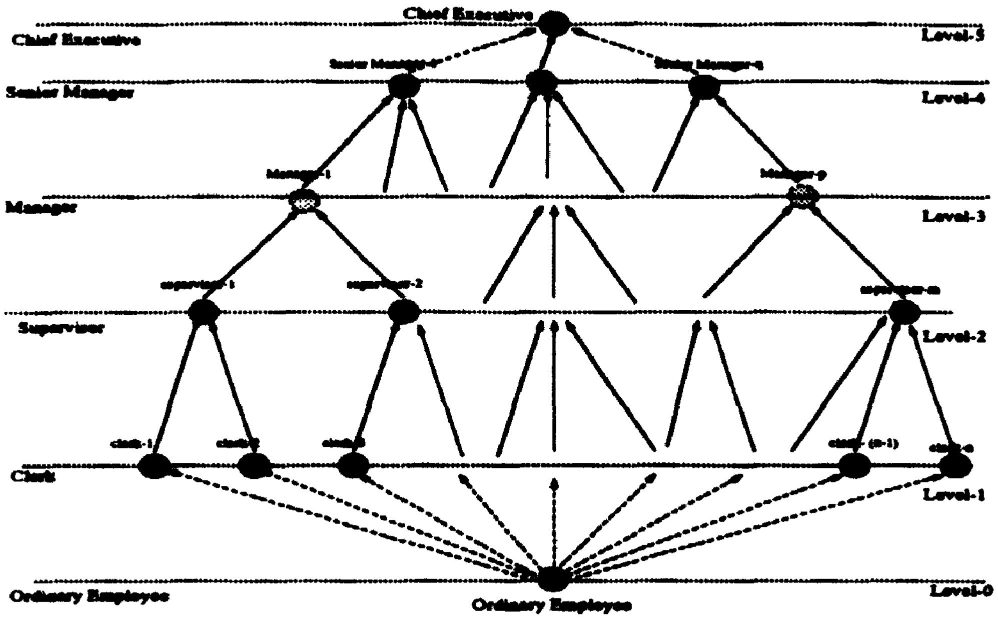

Figure 6.6: Another Example of Role Organization

$r) \wedge (r_j \rightarrow^+ r) \wedge (\not\exists r_l, (r_i \rightarrow^+ r_l \rightarrow^+ r) \wedge (\{r_j \rightarrow^+ r_l \rightarrow^+ r))$ *with both* $(r_i \rightarrow^+ r) \wedge (\cdots r_j \rightarrow^+ r)$ *being the shortest paths of this kind. The lubs, in its most general form, is a set of roles and hence not necessarily a unique role.* □

4. There exists **MinRole** representing the minimum privilege set associated with every role in the system. In general, this *least lower bound* has a relationship of the form MinRole $\rightarrow^+ r$ for any role $r \in \mathcal{R}$. Where there is no common set of privileges for all roles in a system, Ψ(MinRole) = $\emptyset$. It serves to make our model complete such that for any two roles, we can always determine a common junior.

5. There exists **MaxRole** representing the cumulative privilege set associated with the system. In general, this *greatest upper bound* has a relationship of the form $r \rightarrow^+$ MaxRole with any role $r \in \mathcal{R}$. It follows that MinRole $\rightarrow^+$ MaxRole.

Considering paths in the role graph, it is noticeable that paths involving neither MaxRole nor MinRole may be of more interest to us. Consequently, we shall use the following role graph path definition in the subsequent sections.

Definition 6.9 *Role Graph Path: A role graph path, p, is of the form* $r_i \rightarrow r_{i+1} \rightarrow \cdots \rightarrow r_{i+n}, n \geq 0 = r_i \rightarrow^* $ *such that* $r_i \neq$ *MinRole* $\wedge\ r_{i+n} \neq$ *MaxRole associating two* *or more roles in the role graph, RG, connected by the flow relation "$\rightarrow$". Respectively, r_{i+n} and r_i are the highest and lowest ranked roles in this path, p.* □

6.5.3 The Role Graph & Privilege Distribution

One can define a function **PATH** (PATH : $RG \mapsto \mathcal{P}$) which enumerates, among other paths, all the possible paths ($\mathcal{P}$) of a given role organization structure, RG. For such a given **RG** the set $\mathcal{P}$ is finite. The function **PATH** enumerates all the linear extensions (topological orderings) [San88, San89] of a given structure **RG**.[5]

In our model, higher authority roles have more privileges than lower authority ones. In general, the three operators $\rightarrow, \odot$ and $\oplus$ define a subsetting, common subset and common superset privilege relationship, respectively, between roles. We enumerate the cases as follows:

1. Given two roles r_i, r_j with $r_i \rightarrow r_j$ we have $\Psi(r_i) \subseteq \Psi(r_j)$ (subsetting). Property 6.1 asserts that the privilege function increases monotonically with the *is-junior* ($\rightarrow$) relationship along any path. Since this function derives from the *is-junior* relationship, it possesses properties similar to those of *is-junior*. Thus, besides monotonicity, it is also *transitive* and *reflexive*, i.e. $\Psi(r_i) \subseteq \Psi(r_j) \subseteq \Psi(r_k) \Rightarrow \Psi(r_i) \subseteq \Psi(r_k)$ and $\Psi(r_i) \subseteq \Psi(r_i)$, respectively.

   The corollary of this is that given two roles r_i, r_j with $\Psi(r_i) \subseteq \Psi(r_j)$, then we must have some path between them, i.e. $r_i \rightarrow^* r_j$.

2. Given two roles r_i, r_j with $r_i \oplus r_j = \{r_k\}$ we have $(\Psi(r_i) \cup \Psi(r_j) \subseteq \Psi(r), \forall r \in \{r_k\}$. We define $\Psi(r_i \oplus r_j) = \bigcup_{r \in (r_i \oplus r_j)} \Psi(r)$ for any two roles r_i and r_j.

   The function Ψ is *commutative, associative* and *reflexive* for roles related via the *common-senior* ($\oplus$) operator. Hence it follows that $\Psi(r_i \oplus r_j) = \Psi(r_j \oplus r_i)$ (commutativity) and $\Psi(r_i \oplus (r_j \oplus r_k)) = \Psi((r_i \oplus r_j) \oplus r_k)$ (associativity). We also have $r_i \oplus r_i = r_i$ (reflexivity) hence $\Psi(r_i) \cup \Psi(r_i) = \Psi(r_i)$ which is the trivial case of the privileges set of a role being a superset of itself.

3. Given two roles r_i, r_j with $r_i \odot r_j = \{r_k\}$ we have $\forall r \in \{r_k\}, \Psi(r) \subseteq \Psi(r_i) \cap \Psi(r_j)$. We define $\Psi(r_i \odot r_j) = \bigcap_{r \in (r_i \odot r_j)} \Psi(r)$ for any two roles r_i and r_j.

   The function Ψ is *commutative, associative* and *reflexive* for roles related via the *common-junior* ($\odot$) operator. Therefore, $\Psi(r_i \odot r_j) = \Psi(r_j \odot r_i)$ (commutativity)

[5]Sandhu's [San88] $\Gamma(P)$ that enumerates all linear extensions of a partial order P is equivalent to **PATH**

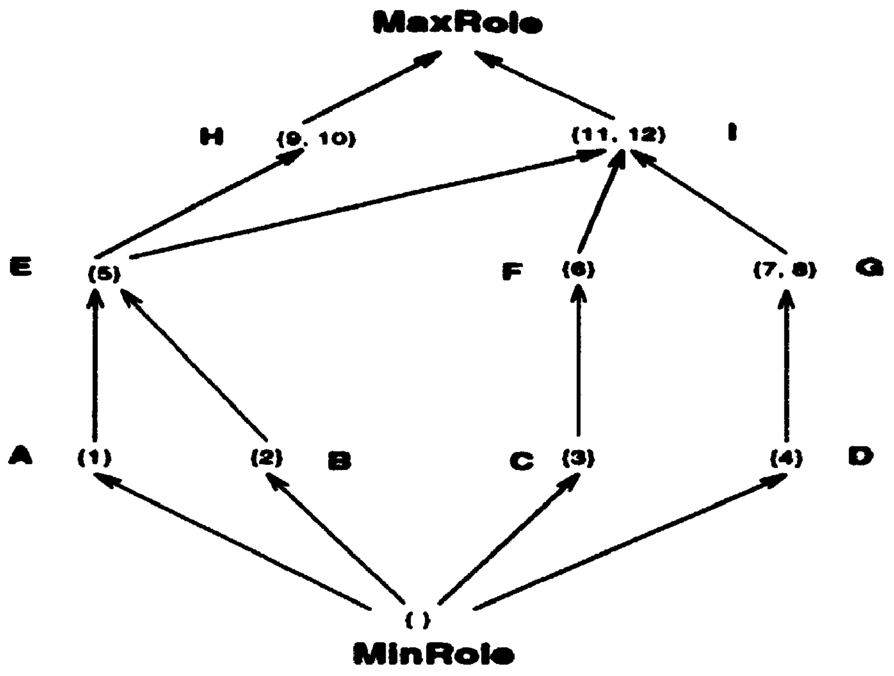

Figure 6.7: Role Graph with Privileges

and $\Psi(r_i \odot (r_j \odot r_k)) = \Psi((r_i \odot r_j) \odot r_k)$ (transitivity). As well, we have, $r_i \odot r_i = r_i$ (reflexivity) we have $\Psi(r_i) \cap \Psi(r_i) = \Psi(r_i)$. This is the trivial case of the privileges set of a role being a subset of itself.

4. Observe the significance of privileges associated with **MaxRole** and **MinRole**. We have $\bigcup_{r \in \mathcal{R}} \Psi(r) \subseteq \Psi(\text{MaxRole})$ while $\bigcap_{r \in \mathcal{R}} \Psi(r) = \Psi(\text{MinRole})$. The former signifies the cumulative privileges that can be exercised in the system while the latter denotes the least privileges at the disposal of any role in the role graph. In terms of real life organizations, **MaxRole** denotes the cumulative authority of a system while **MinRole** denotes the very least authority in the system to which everyone is entitled.

Definition 6.10 *Path Role Set: The role set of a given path is the set of all roles that compose the path. We say that a given role participates in a path if it belongs to the path's role set. It is given by $\Gamma(p)$ where Γ is defined as $\Gamma : \mathcal{P} \mapsto \mathcal{R}$. $\mathcal{P}$ and $\mathcal{R}$ are the universal path and role sets in the system.* □

Definition 6.11 *Path Independence: Let $\Gamma(p_i)$ and $\Gamma(p_j)$ be the role sets of paths p_i and p_j, respectively. We say p_i is independent of p_j if and only if and only if $\forall r_i \in \Gamma(p_i), \forall r_j \in \Gamma(p_j), r_i \oplus r_j = \{MaxRole\}$ and $r_i \odot r_j = \{MinRole\}$. Alternatively, p_i is independent of p_j if and only iff $\Psi(\Gamma(p_i) \cap \Gamma(p_j)) = \Psi(MinRole)$.* □

Privilege Distribution Table For Figure 6.7			
Role Name	Direct (D)	Indirect (I)	Effective ($D \cup I$)
A	{1}	{ }	{1}
B	{2}	{ }	{2}
C	{3}	{ }	{3}
D	{4}	{ }	{1}
E	{5}	{1,2}	{1,2,5}
F	{6}	{3}	{3,6}
G	{7,8}	{4}	{4,7,8}
H	{9,10}	{1,2,5}	{1,2,5,9,10}
I	{11,12}	{1,2,3,4,5,6,7,8}	{1,2,3,4,5,6,7,8,11,12}

Table 6.1: Table of Privileges

In other words, the two role sets are not related other than via MaxRole and MinRole. Such independence, as we shall see later, guarantees that independent paths do not share privileges.

Concerning privilege distribution, the privileges directly assigned to a role are termed *direct* while those *implicitly* acquired from roles in its junior role set are termed *indirect*. For a given role r, *Direct*($\Psi(r)$) denotes its direct privileges. The effective privilege set of a role r, which we have been denoting by $\Psi(r)$, is the union of its direct and indirect component privileges. This relationship between a node's junior role set and its privilege set is important and eases the task of system privilege administration. It determines the privileges available to users via the role associated with a node for users authorized for the role.

Example 6.2 **Consider figure 6.7 where we have distinct privileges numbered $1, \cdots, 12$ with privileges $1, \cdots, 6$ directly assigned to roles $A, \cdots, F$ and $\{7,8\}, \{9,10\}, \{11,12\}$ assigned roles G, H, I respectively. We have a role graph specification as follows: $A \rightarrow E, B \rightarrow E, C \rightarrow F, D \rightarrow G, E \rightarrow \{H, I\}, \{F, G\} \rightarrow I$, MaxRole = $H \oplus I$, and MinRole = $A \odot B \odot C \odot D$ with Ψ(MinRole) = $\emptyset$.**

From this we can compute the privileges of various roles and obtain the privileges distribution as in table 6.1. Moreover, we have the following relationships relating to the $\odot, \oplus, \rightarrow$ operators:

1. **The *common-junior* operator, $\odot$, defines a common subset of privileges for any two roles. Consider $H \odot I = E$ and note that $\Psi(H \odot I) = \Psi(E) = \{1,2,5\} = \Psi(H) \cap \Psi(I)$.**
2. **The *common-senior* operator, $\oplus$, defines the union of privileges of two roles and as**

such is a common superset for any two roles. Consider $F \oplus G = I$ and note that $\Psi(F \oplus G) = \Psi(F) \cup \Psi(G) = \{3,4,6,7,8\} \subseteq \Psi(I) = \{3,4,6,7,8,11,12\}$.

3. **The *is-junior* operator, $\rightarrow$, defines a proper subset relationship between two roles, e.g. $E \rightarrow H$. Note that $\Psi(E) = \{1,2,5\} \subset \{1,2,5,9,10\}$. This is true for all roles related via the *is-junior* relationship.**
4. **Paths $A \rightarrow E \rightarrow H$ and $C \rightarrow F$ are independent paths since their roles sets {A,E,H} and {C,F} are mutually exclusive and the two paths are related via only via MaxRole and MinRole.**

□

6.5.4 Role Organization Model & Operator Semantics

The structure $RG = (\mathcal{R}, \rightarrow)$ is role graph which characterized by it its four components $(\mathcal{R}, \rightarrow, \oplus, \odot)$ as well as **MaxRole** and **MinRole**. Collectively, they can be viewed as defining an **authority structure** for the roles. The relation $r_i \rightarrow r_j$ means that r_i is subservient to role r_j, i.e. authority increases along the "$\rightarrow$" relation. For any **role path** of the form $r_i \rightarrow \cdots \rightarrow r_n, n \geq 1$ we have an authority relation of the form $r_1 < \cdots < r_n$ with the roles totally ordered. In general, given any two roles $r_1, r_2 \in \mathcal{R}$, $r_1 < r_2$, $r_2 < r_1$ or they are incomparable. Hence their partial ordering. Where there is a path (call it an **authority flow path**) the roles in the path form a total order. Such a path is essentially a linear ordering, also termed a linear extension in [San88, San89], of a subset of a set of roles in the organization structure **RG**.

6.5.5 The Model Properties

Besides acyclicity and monotonicity, other interesting properties of this model include the fact that any node in the model has the same properties as the overall structure. Moreover, any role can be decomposed into any number of sub-roles while at the same time retaining the *mandatory role organization property* (see property 6.5).

Property 6.5 Mandatory Role Graph Property: *Any node in the role graph model is itself a role graph.* □

The structure can be "expanded" at any point by adding new roles as the need arises while retaining the mandatory role structure property, a strategy that offers a flexible manner of introducing new privilege into the model. Such privilege can be incorporated in an existing structure by introducing new roles or by increasing the privilege of existing ones.

The set of privileges of any role is the union of all privileges of its associated junior roles. This leads to another key property:

Property 6.6 Role Graph Privilege Set Invariant Property: *The privilege set of a given role graph (or a subgraph of a role graph) remains invariant unless altered through some system security function.* □

6.5.6 Other Role Graph Characterizations

Other important characterizations of the role organization model can be formulated. Considering the role organization model proposed in section 6.5.2, we term the extent of *linkage* between roles a *coupling* which is related to the extent to which privileges are shared among roles. We can have a variety of cases, e.g. where each role is independent of all others or where some roles are *coupled* and hence dependent on each other. To characterize these facts we introduce the concepts of the *role coupling set* and *role coupling factor* between sets of roles. We define these two in terms of shared roles among the role set, i.e. roles related via the $\odot$ operator. Before that we formally define coupling between two roles:

Definition 6.12 Coupling: *Coupling exists between two roles r_i and r_j if $\exists\{r_k\}$ such that $r_i \odot r_j = \{r_k\} \wedge \{r_k\} \neq \{MinRole\}$. We call $\{r_k\}$ the coupling between r_i and r_j. The coupling between two or more roles the greatest lower bound set of the roles.* □

Independent roles have no coupling between them, i.e.

Definition 6.13 Role Independence: *Two roles r_i and r_j are independent if and only if $r_i \odot r_j = \{MinRole\}$, i.e. their only coupling is the role common to all roles in the structure. In other words their greatest lower bound is the MinRole.* □

As we shall see later in the next chapter, this independence is important is ensuring there is no conflict of interest between different roles when assigning roles to users.

Notice that there is a very close relationship between role coupling and role independence. This arises from that fact that each role can be decomposed into any number of roles provided the privilege set invariant property is retained.

We define the *coupling set* between two roles as the role set associated with the coupling role of the two roles. Formally:

Definition 6.14 Role Coupling Set: *The coupling set of two roles r_i and r_j is $CS(r_i, r_j) = \{r_k\}$ such that $\forall r \in \{r_k\}, (\Psi(r) \subseteq \Psi(r_i)) \wedge (\Psi(r) \subseteq \Psi(r_j)) \wedge (r \neq MinRole)$ The cou-*

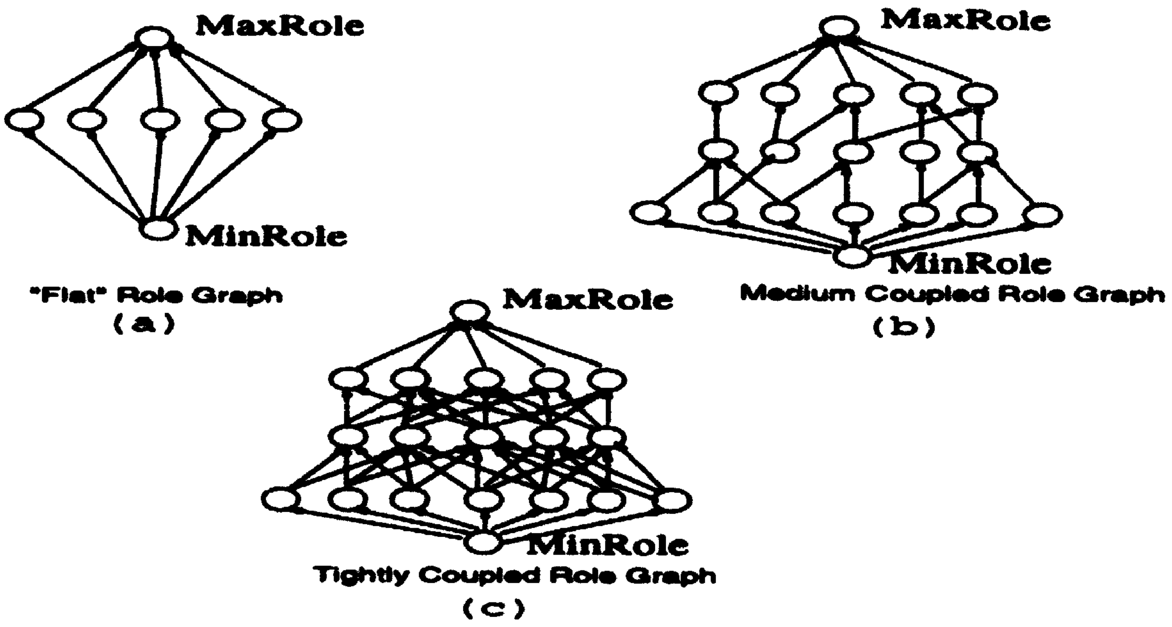

Figure 6.8: Different Levels Coupling Role Graphs

pling set of a single role r_i is the set $\{r_k\}$ such that $\forall r \in \{r_k\}, ((\Psi(r) \subseteq \Psi(r_i)) \wedge (r \neq MinRole)$, i.e. all roles with a common-junior relationship with the role. □

Role Independence is also closely related with *path independence*, where

Definition 6.15 *<u>Independent Paths:</u> Two paths are independent if the coupling set between their role sets is empty, i.e. there are no shared roles between the roles that compose the two paths. In other words, the two paths are independent when every role from one path has empty coupling set with every roles from the other path and vice versa (see also definition 6.11).* □

Definition 6.16 *<u>The Coupling Factor:</u>*
The coupling factor CF of two given roles r_i and r_j is the cardinality of their coupling set, i.e. $CF(r_i, r_j) = |CS(r_i, r_j)|$ or simply $CF = |CS|$. CF for a single role follows from the definition of CS for a single role. □

Notice that $0 \leq$ CF $\leq CF(MaxRole)$, i.e. the size of the coupling set is zero where there is no coupling at all and is equal to the number of roles in the system (except MinRole) where there is maximum role coupling. Where the CF involves two or more roles, in the former, the roles are said to be *independent* while in the latter the roles completely interwoven and hence they have maximum dependency.

In talking about coupling, it makes sense to talk about coupling of roles other than MaxRole and MinRole. Hence, without considering these two, we say a role organization structure can be *loosely* coupled, *tightly* coupled or not coupled at all. In the former, two or more roles in the systems would have a low coupling factor while similar roles would have a higher coupling factor, in the latter. A structure without coupling has a coupling factor equal to zero for many of its roles, a loosely coupled one has a coupling factor close to zero while a tightly coupled structure has a coupling factor that tends to the maximum value of the coupling factor. In a structure without coupling all paths are independent. Where there is coupling we have no guarantee that there will be independent paths.

The desirable extent of role coupling would be a factor of the enforced security policy and the attendant ease of privilege administration it offers. This extent of coupling would, inevitably, be related to the flexibility of role privilege administration and consistency and completeness analysis.

We identify two extreme forms: the "flat single-level" structure[6] and the multilevel tightly inter-related forms (see figures 6.8b and 6.8c, respectively). In the former, each role is independent of all others except for the special MinRoles and MaxRoles. Determining the privileges associated with a role is a straight forward task in which we get the privileges *explicitly* associated with the role in question. A simple enumeration is sufficient for such a task. In the latter case, we have each role coupled (via the $\rightarrow$ operator in the model of section 6.5.2) with two or more other roles. Thus we have privilege sharing determined by the extent of coupling within the structure. Determining the privilege associated with a particular role requires determining the junior role set of the role and taking the union of all the privileges in the junior role set. This case clearly involves some implicit step in which *indirectly* assigned privileges are determined.

Midway between these "extremes" (see figure 6.8b for example) lie a range of cases with varying levels of coupling hence varying levels of complexity in determining the privilege associated with some role. Indeed, we can have differing levels of "horizontal" and "vertical" spread of the role organization structure.

[6]Recall that each role is a role organization structure and with vertical partition we can turn this structure into independent paths with more than one role each. The term "flat single-level structure" is used loosely!

6.6 The Role Graph & Privilege Administration

The major benefit of formulating the role graph is to facilitate the ease of access rights administration. It is thus important that the manner in which the privileges are distributed throughout the role graph be *optimized*. To do so involves taking into account the role relationships specified as well as those inferred from privilege relationships. Relationships specified are of the form $r_i \rightarrow r_j, r_i \oplus r_j = \{r_k\}, r_i \odot r_j = \{r_k\}$. Relationships inferred result from the analysis of privileges associated with the given roles. For instance, if we find that the privileges of one role form a subset of those of another, then we can *infer* an *is-junior* relationship between the two. This then implies a *mandatory path* requirement whenever there is a subsetting privilege relationship. There must exist a path between any two roles with *subset* privilege association. This requirement aims at reducing privilege distribution redundancy and this forms the basis of privilege distribution optimality. Both privilege distribution optimization and transitive reduction [AGU72] (see below) characterize role graph *well-formedness.*

A role graph is a role relationship structure derived from privilege relationships (specified and inferred) among roles. It can thus be seen as a privilege distribution scheme. It is important that privilege duplication within roles, itself a form of redundancy, be minimized by taking into account the privilege relationships between roles. A privilege distribution scheme is said to be *optimal* if privilege duplication within roles is minimized. In other words, considering the direct privileges defined in roles, duplication should be minimized by taking into account the role relationships on which basis the role graph is formulated. This has the effect of minimizing privilege distribution redundancy where privilege redundancy is defined as:[7]

Definition 6.17 *Privilege Distribution Redundancy: There is privilege distribution redundancy between any two roles r_i, r_j whenever we have $r_i \rightarrow^+ r_j$ and $\Psi(r_i) \cap Direct(\Psi(r_j)) \neq \emptyset$.* □

Consider a role graph with two roles X and Y. Suppose it is given that: MinRole $\rightarrow X, X \rightarrow$ MaxRole, MinRole $\rightarrow Y$ and $Y \rightarrow$ MaxRole. Without knowledge of the privileges associated with the two roles X and Y, no other relationship information can be deduced. As such it is correct to assume that the two roles are independent.

[7]Recall that $Direct(\Psi(r))$ and $\Psi(r)$ refer to the direct and effective privileges, respectively.

The role graph would then have paths MinRole $\rightarrow X \rightarrow$ MaxRole and MinRole $\rightarrow$ $Y \rightarrow$ MaxRole.

By examining the associated privileges, however, it is possible to determine whether the two are related. For instance suppose that $\Psi(X) = \{1\}$ and $\Psi(Y) = \{1,2\}$. It follows that $\Psi(X) \subseteq \Psi(Y) \Rightarrow X \rightarrow Y$. We can then have $Direct(\Psi(Y)) = \{2\}, Direct(\Psi(X)) = \{1\}$ and $X \rightarrow Y$. Once we know this, privilege 1 need not be duplicated in the direct privileges of Y. It can be inferred from the relationship between X and Y. This then reduces the role graph to only one path: MinRole $\rightarrow$ $X \rightarrow Y \rightarrow$ MaxRole.

The mandatory path requirement for roles related via the subsetting privilege relationship, as a property of the role graph, is expressed as follows:

Property 6.7 *Mandatory Path: A path $r_i \rightarrow^* r_j$ must exist between any two roles r_i, r_j whenever $\Psi(r_i) \subseteq \Psi(r_j)$.* □

A role graph in which all roles with *is-junior* relationship are connected by some path and in which there is an *is-junior* relationship whenever there is a privilege subsetting relationship, is said to be in *proper form.* Formally:

Definition 6.18 *Role Graph Proper Form: A role graph is in proper form if and only if (1) there is some path between all roles with an* is-junior *relationships and (2) there is an inferred* is-junior *relationship between two roles whenever there is a subset relationship between their privilege sets.* □

A graph in proper form captures *all possible* paths. A role graph in proper form can be used to eliminate the redundancy in the privileges associated with a role in the graph. The resulting role graph with privilege redundancy removed is said to realize privilege distribution optimality.

Definition 6.19 *Privilege Distribution Optimality: A privilege distribution scheme is optimal if and only if (1) it captures all possible (specified and inferred) paths and (2) there is no redundancy in privilege distribution in the graph, i.e. for any roles (r_i and r_j) with a path $r_i \rightarrow^+ r_j$ between them, $Direct(\Psi(r_j)) \cap \Psi(r_i) = \emptyset$.* □

This definition implies that all privileges defined for r_i must not be *duplicated* in r_j whenever we have $r_i \rightarrow^+ r_j$. This process ensures that we retain a privilege in a role's direct privilege set if and only if it has *not* already been specified in the associated junior roles of the role in question. In other words, all direct privileges of a given role r_j should be those such that there exists no role in the role's junior set in which the privilege has been defined. Hence in our example from above, we shall have only privilege "2" directly defined in Y since privilege "1" is already defined in X which has an *is-junior* relationship with Y.

The above process can rightly be termed ***privilege distribution optimization.*** It is the process by which privilege duplication is minimized. It aims at reducing privilege distribution redundancy.

The foregoing addresses redundancy elimination via minimization of privilege duplication within roles in a role graph. However, it does not address another form of redundancy: ***path redundancy.***

Definition 6.20 Path Redundancy: *There exists path redundancy whenever we have paths of the form: $r_i \rightarrow^+ r_j \rightarrow^+ r_k$ and $r_i \rightarrow r_k$ simultaneously.* □

Path redundancy can be eliminated via ***transitive reduction*** [AGU72] which is a process by which, given paths of the form: $r_i \rightarrow^+ r_j \rightarrow^+ r_k$ and $r_i \rightarrow r_k$, the redundant edge $r_i \rightarrow r_k$ is removed. This process, when applied to a directed graph, removes all redundant edges while leaving all the relationships pertaining to the graph intact. The resulting graph is said to have been transitively reduced; it is a transitive reduction.

A graph in which there is neither redundant path nor redundant privilege distribution is said to be *well-formed.*

Definition 6.21 Role Graph Well-Formedness: *A role graph is well-formed if it is a transitive reduction and if its privilege distribution is optimal.* □

It is important that a role graph, at any one time, be well-formed. Any operations that transform the graph assume a well-formed graph as input and must assure to leave it in a well-formed state.

There are other interesting modifications that can be done on a role graph. These include the reduction and/or addition of role privileges which requires the specification of the target role and privileges to be removed/added but which does not alter the basic structure and relationships in the role graph structure. This may be addressed within the context of role-privilege authorization. However, these modifications should preserve privilege distribution optimality.

6.7 The Role Graph & Role Administration

In this section we briefly outline role administration in the proposed role graph model and demonstrate the soundness of the operations suggested . Included in the task of role administration are operations of *role deletion* (e.g. when that role is no longer required), *role addition* (when there arises a need to create a new role), or *role partition*, vertical or horizontal (when there is a need to partition the functionality of the role). In all these cases, we can have an increment or decrement in the overall privileges associated with roles related to the one undergoing change. Such privileges may be kept either invariant, reduced or increased depending on the specified operations. We address the cases where (1) path privileges are introduced with the addition of a new role, (2) path privileges remain invariant with the deletion of a role, (3) path privileges are "partitioned" with the partition (horizontal partition) of a role and (4) path privileges remain invariant with the partition (vertical partition) of a role.

In each of these cases we assume a well-formed role graph (see section 6.6), i.e. one in which privilege distribution is *optimal* and to which transitive reduction [AGU72] has been done. The resulting graph after the operation is guaranteed to be well-formed.

Consequently, after carrying out the operations on the graph, our procedures will confine themselves with the immediate neighbourhood of the target role. Such a "neighbourhood" can be seen as the roles along the paths affected by the operation. Hence in doing privilege optimization, we look at the paths in which privilege changes have taken place while with transitive reduction we examine the affected paths for any redundant arcs. The following definitions will be useful with respect to a role's "neighbourhood":

Definition 6.22 *Juniors(r): The set Junior(r) for a given role r is all $r_i \in \mathcal{R}$ such that $r_i \rightarrow^+ r$.* □

Juniors(r) can be seen as the set of roles in the *subgraph* associated with the role r without the role itself.

Similarly·

Definition 6.23 *Seniors(r): For a given role r, the set Seniors(r) is all $r_i \in \mathcal{R}$ such that $r \rightarrow^+ r_i$.* □

As with the *Juniors(r)*, *Seniors(r)* can be seen as the subgraph defined by r minus the role, after inverting the role graph.

6.7.1 Role Addition & Deletion

Role addition means the creation and incorporation of a totally new role in the role graph. Such a role would be defined (name and privilege list) before being integrated into the role graph. While the integration process must preserve the role definition, it is important to ensure that if there are privileges defined in the new role that exist in junior roles in the target paths, they must be removed to take away the redundancy. To introduce such a role requires the specification of the target path and the position in the path. Such a position is determined by defining the immediate superior and subservient roles to be related to the new role. We term these superior and junior lists, respectively. See algorithm 6.1 in figure 6.9 and also figure 6.10b.

Example 6.3 **With role addition we specify the target path(s) and points of insertion. This involves the specification of the target superior and junior role(s) for the role to be added (see figure 6.10a). The role to be inserted is added with the appropriate links being specified to indicate the junior and superior roles. Finally redundant paths are removed from the resulting structure.** □

The flip side of *role addition* is *role deletion* which involves the elimination of a role from the role organization structure. Th. process requires specifying the target role and *short-circuiting* it by making the target's immediate subservient role the immediate subservient roles of the target's immediate superiors. In doing so, the privileges associated with the deleted role can either be eliminated or distributed. This must be specified in the role deletion process. Privilege elimination involves overall privilege reduction of the path associated with the role so deleted.

Retaining the privileges of the deleted role, on the other hand, requires a specification of how these privileges will be distributed among the existing roles. Usually, such privileges would be distributed within the immediate neighbourhood of the deleted role.

In deleting a role, it must be specified whether the privileges will be deleted with the role or whether they will be distributed among the existing roles. For instance some may be re-allocated to the immediate superior role while others may be allocated to the immediate junior role. In practice, however, if the task is to keep the privilege set of the paths invariant, the task would be easier if the privileges are allocated to the superior role. When privileges of a role are distributed, the overall effect is that of ordering the privileges and total path privileges may remain the unchanged. This case is illustrated pictorially in figure 6.11. See the associated algorithm 6.2 of

Algorithm 6.1 Role_Addition(rg, target, s.target.set j.target.set)

```
/* For the addition of a given role into a role graph */
Input: $rg = \langle \mathcal{R}, \rightarrow \rangle$ (role graph), target to be added (name with proposed direct privilege set),
s.target.set (immediate superior set for target), j.target.set (immediate junior set for target).
Output: The role graph with target added and overall privileges of other roles left intact.
Var $r, r_j, r_s$: roles;
Begin
    If $\exists(r_s \rightarrow^+ r_j)$ for any $r_s \in s.target.set, r_j \in j.target.set$
        Then abort                                   /* Must not violate acyclicity */
    Else Begin
        1. $\Psi$(target):= $(\cup_{r \in j.target.set} \Psi(r)) \cup Direct(target)$;
                /* Compute the effective privileges of target role */
        2. If $\Psi(r) = \Psi(target)$ for any $r \in \mathcal{R}$
             Then $target := r$;          /* Role privilege sets must be unique */
        3. If $\exists(target \rightarrow^+ r_j)$ for any $r_j \in j.target.set$
             Then abort                              /* Must not violate acyclicity */
          Else Begin
             a. $\mathcal{R} := \mathcal{R} \cup target$;          /* Add target to system roles */
             b. $\forall r_s \in s.target.set$ do add the edge $target \rightarrow r_s$;
             c. $\forall r_j \in j.target.set$ do add the edge $r_j \rightarrow target$;
             d. If for any $r \in \mathcal{R}, \Psi(r) \subset \Psi(target)$ and NOT($r \rightarrow^+ target$)
                    Then add the edge $r \rightarrow target$;    /* Add this inferred edge */
             e. If for any $r \in \mathcal{R}, \Psi(target) \subset \Psi(r)$ and NOT($target \rightarrow^+ r$)
                    Then add the edge $target \rightarrow r$;    /* Add this inferred edge */
             f. Rem_Red_Arcs(rg, j.target.set, s.target.set, target);
             g. Red_Priv_Res(rg, j.target.set, target); end;
        4. $\forall r_i, r_j \in \mathcal{R}$ do if $\Psi(r_i) = \Psi(r_j)$ then        /* Remove duplicate roles */
           Begin $\forall r$ such that $r_i \rightarrow r$ do add the edge $r_j \rightarrow r$
             $\forall r$ such that $r \rightarrow r_i$ do add the edge $r \rightarrow r_j$
             Delete all edges $r_i \rightarrow r$ and $r \rightarrow r_i$;
             Remove $r_i$; end;
        5. $\forall r_i, r_j, r_k \in \mathcal{R}$ do if $\exists((r_i \rightarrow^+ r_j \rightarrow^+ r_k)$        /* Remove redundant edges */
           $\wedge (r_i \rightarrow r_k))$ then
             Delete the edge $r_i \rightarrow r_k$,end;              /* Delete the direct edge */
end.                                                  /* Role_Addition */
Procedure Rem_Red_Arcs(var rg: role graph; j.target.set, s.target.set: role_set; target: role );
/* Removes redundant arcs in the immediate neighbourhood of target role */
Var $r_k, r_j, r_s$: roles;
Begin 1.  $\forall r_j \in j.target.set$ do            /* Remove direct paths */
          if $\exists(r_j \rightarrow r_k \rightarrow \cdots \rightarrow target)$ then
             Delete the edge $r_j \rightarrow target$    /* Delete the direct edge*/
      2.  $\forall r_s \in s.target.set$ do
          if $\exists(target \rightarrow r_k \rightarrow \cdots \rightarrow r_s)$ then
             Delete the edge $target \rightarrow r_s$    /* Delete the direct edge*/
end;                                        /* Red_Red_Arcs */
Procedure Red_Priv_Res(var rg: role graph; j.target.set: role_set; target: role);
Var pv : privilege; r : role;
Begin 1. For all r in Seniors(r) do          /* Remove redundant privileges in seniors. */
          $\forall pv \in Direct(target)$ do
            if $pv \in Direct(r)$ then
               $Direct(r) := Direct(r) - pv$;
      2. For all r in j.target.set do        /* Remove redundant privileges in Direct(target).
          $\forall pv \in \Psi(r)$ do
            if $pv \in Direct(target)$ then
               $Direct(target) := Direct(target) - pv$;
end;                                        /* Red_Priv_Res */
```

Figure 6.9: Algorithm for Role Addition

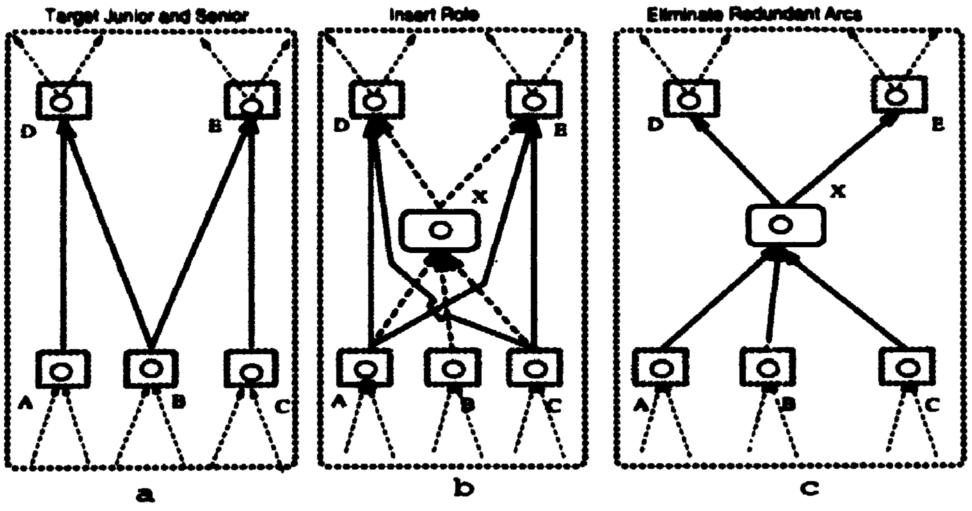

Figure 6.10: Role Addition

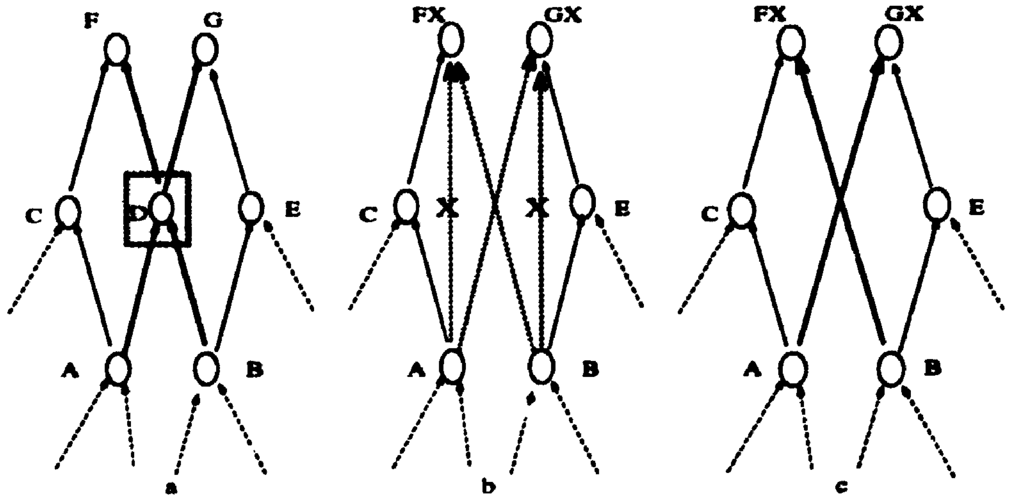

Figure 6.11: Role Deletion

figure 6.12.

Example 6.4 **Suppose our target role for deletion is role D in figure 6.11a with the constraint that all existing paths must keep their privilege sets invariant. For this purpose we choose to shift the privilege set of the target role to its superiors.**

To achieve this, first transfer the privileges from role D to both F and G which are both superior to D. This results in roles roles FX and GX which we make immediate superiors of both A and B which were immediate juniors of role D. The graph retains all previous relations, i.e. $A \rightarrow D \rightarrow F, A \rightarrow D \rightarrow G, B \rightarrow D \rightarrow F$, **and** $B \rightarrow D \rightarrow G$ **are replaced by** $A \rightarrow FX, A \rightarrow GX, B \rightarrow FX$, **and** $B \rightarrow GX$**, respectively.**

The next step is to do away with redundant alternative paths (marked X in the figure 6.11b) and remove them. We notice that paths $A \rightarrow C \rightarrow FX$ **and** $B \rightarrow E \rightarrow GX$ **contain the set of privileges of paths** $A \rightarrow FX$ **and** $B \rightarrow GX$**, respectively. This results in a new role graph structure as shown in figure 6.11c.**

We consider this as representative of the most general case as it involves more than one path. Where the target role involves just one path, the task should be by far easier than that outline here. □

Both role addition and deletion correspond to real life situations where in creating a new portfolio, a new role is added, while in eliminating some "office", a role will be deleted. Role deletion without privilege reduction entails elimination of some "office" in an organization while retaining the total functionality of the organization. Indeed, this may be the usual practice.

6.7.2 Role Partition

A role can be partitioned into two or more roles in our role graph. Essentially, the basic partition operations are either vertical or horizontal, and can of course be combined. In both cases it must be specified what the new roles and their corresponding privileges are. Where the order of "seniority" is required, as in the case of vertical partition, it must be specified as well.

In **vertical role partition**, a role is split into two or more roles and an ordering is imposed on them with the *is-junior* relationship. In doing vertical partition, we must specify the target role, the new roles to be created, their direct privileges and their ordering (according to partial privilege criterion). For instance, a role X is not only partitioned into roles $X_1, \cdots, X_n$ but also, these roles must be ordered, e.g. $X_1 \rightarrow \cdots \rightarrow X_n$ (see figure 6.13b and algorithm 6.3 of figure 6.14). Privilege distribution among the new roles is constrained by the privileges associated with the

Algorithm 6.2 Role_Deletion(rg, target, inv)

```
/* Deletes a specified role retaining or discarding its privileges depending on inv */
Input: $rg = \langle \mathcal{R}, \rightarrow \rangle$ (the role graph structure), target (the target role to be deleted),
        inv Boolean indicating whether or not to retain the role's privileges
Output: The role graph structure with target deleted

Var s.set, j.set: role set; $r, r_j, r_s$: role;
        Begin 1. s.set := Superior_Set(target);        /* Get the senior set */

              2. j.set := Junior_Set(target);          /* Get the junior set */
              3. For all $r_s \in s.set$ do                /* Connect Junior and Senior Roles */
                    For all $r_j \in j.set$ do add $r_j \rightarrow r_s$;
              4. If inv then do
                    For all $r_s \in s.set$ do             /* Transfer Privileges to superiors */
                       $Direct(r_s) := Direct(r_s) \cup Direct(target)$;
              5. For all $r_s \in s.set$ do                /* Remove all redundant arcs */
                    For all $r_j \in j.set$ do
                       If $\exists(r_j \rightarrow r_k \cdots \rightarrow r_s)$ then delete $r_j \rightarrow r_s$;
              6. $\mathcal{R} := \mathcal{R} - target$;              /* Take out target from system roles
        end.                                           /* Role_Deletion */

        Function Superior_Set(var rg: role graph; target: role): role_set;
        Var Tempset: role_set; r: role;
        Begin 1. Tempset := $\emptyset$;
              2. For all r with $target \rightarrow r$ do
                    Tempset := Tempset $\cup$ r;
              3. Superior_Set := Tempset
        end;                                           /* Superior_Set */

        Function Junior_Set(var rg: role graph; target: role): role_set;
        Var Tempset: role_set; r: role;
        Begin 1. Tempset := $\emptyset$;
              2. For all r with $r \rightarrow target$ do
                    Tempset := Tempset $\cup$ r;
              3. Junior_Set := Tempset
        end;                                           /* Junior_Set */
```

Figure 6.12: Algorithm for Role Deletion

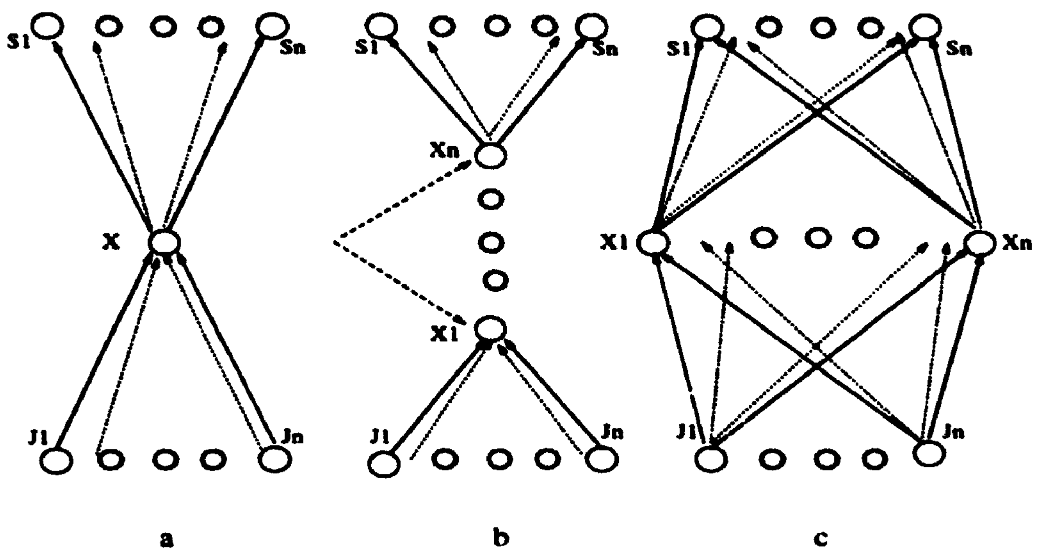

Figure 6.13: Vertical & Horizontal Role Split

role being partitioned; there *must not* be an increment or decrement of privileges, i.e.

$$Direct(X) = \bigcup_{i=1,\cdots,n} Direct(X_i)$$

Consequently, the privileges associated with the paths in which the role appears neither decrease nor increase. In general, vertical partition leaves the privilege set associated with all paths unaffected; only the path length increases.

Further constraints include the requirement for distinct direct privilege sets for the newly created roles, i.e. for any

$$X_i, X_j \in \{X_1, \cdots, X_n\}, Direct(X_i) \bigcap Direct(X_j) = \emptyset$$

Suppose we have a target role for partition (call it X) with a relationship $\{J_1, \cdots, J_n\} \rightarrow X \rightarrow \{S_1, \cdots, S_n\}$ which is partitioned vertically into roles $\{X_1, \cdots, X_n\}$ such that $\{J_1, \cdots, J_n\} \rightarrow \{X_1 \rightarrow \cdots \rightarrow X_n\} \rightarrow \{S_1, \cdots, S_n\}$. It follows that $(X_n \subseteq (S_1 \oplus S_2 \oplus \cdots \oplus S_n)) \wedge (X_1 \subseteq (J_1 \oplus J_2 \oplus \cdots \oplus J_n))$.

Horizontal role partition, on the other hand, involves partitioning a role into two or more roles with none of them being subservient (superior) to another (see figure 6.13c and algorithm 6.4 of figure 6.15). Partition, as used here, merely distributes the direct privileges of the target role among newly created roles that replace it. In partitioning a role, there should be no effective increment or decrement of privileges. In other words, as with vertical partitioning, if role X is partitioned into

Algorithm 6.3 Vertical_Partition(rg, target, $\{((x_i, \rightarrow), x_i.rpset_i)\}$)

```
/* Partitions a given role vertically */
Input: $rg = \langle \mathcal{R}, \rightarrow \rangle$ (the role organization structure), target (the target role to be partitioned),
    $\{((x_i, \rightarrow), rpset_i)\}$ (the new role-direct privilege set pairs and their ordering).
Output: The role graph with target vertically partitioned into $\{((x_i, \rightarrow), rpset_i)\}$ and
    integrated into the role graph structure.
Uses Superior_Set and Junior_Set of algorithm 6.2 in figure 6.12.

Var s.set, j.set: role set; $r_j, r_s$: roles;

    Begin
        If $Direct(target) \neq \bigcup(Direct(x_i))$
            Then abort /* Must keep privilege set invariant */
        Else Begin
            1. $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;                /* Add new roles to system */
            2. s.set := Superior_Set(target);   /* Generate superior set */
            3. j.set := Junior_Set(target);     /* Generate Junior set */
            4. Add edges $x_1 \rightarrow x_2, x_2 \rightarrow x_3, \cdots, x_{n-1} \rightarrow x_n$;
                                                /* Create a Path as specified */
            5. $\forall x_i$ do $Direct(x_i) := rpset_i$;   /* Assign the appropriate privileges */
            6. $\forall r_s \in s.set$ do add $x_n \rightarrow r_s$;   /* Join the Senior end */
            7. $\forall r_j \in j.set$ do add $r_j \rightarrow x_1$;   /* Join the Junior end */
            8. $\mathcal{R} := \mathcal{R} - target$;            /* Delete target from system */
        end;
    end.                                        /* Vertical_Partition */
```

Figure 6.14: Algorithm for Vertical Partition

roles $X_1, \cdots, X_n$, we require that

$$Direct(X) = \bigcup_{i=1,\cdots,n} Direct(X_i)$$

The direct privilege sets of these newly created roles can have empty or non-empty intersections. However, none of them should have identical privilege sets. Note that, unlike vertical partition, horizontal partition can cause a variation of privileges associated with a path when the target role is the senior-most role in the path.

Suppose we have a target role for partition (call it X) with a relationship $\{J_1, \cdots, J_n\} \rightarrow X \rightarrow \{S_1, \cdots, S_n\}$ which is partitioned horizontally into roles $\{X_1, \cdots, X_n\}$ such that $\{J_1, \cdots, J_n\} \rightarrow \{X_1, \cdots, X_n\} \rightarrow \{S_1, \cdots, S_n\}$. It follows that $(\{J_1, \cdots, J_n\} \subseteq (X_1 \odot X_2 \odot \cdots \odot X_n)) \wedge (\{S_1, \cdots, S_n\} \subseteq (X_1 \oplus X_2 \oplus \cdots \oplus X_n))$

Updates to the role graph include the reduction and addition of role privileges which require the specification of the target role and privileges to be removed/added,

Algorithm 6.4 <u>Horizontal_Partition(rg, target, $\{(x_i, x_i.rpset_i)\}$</u>

```
/* Partitions a given role horizontally */
Input: $rg = (\mathcal{R}, \rightarrow)$ (the role organization structure), $target$ (the target role to be partitioned),
    $\{(x_i, rpset_i)\}$ (the new role-direct privilege pairs to replace $target$).
Output: The role structure with $target$ horizontally partitioned into $\{(x_i)\}$ and integrated into $rg$.
Uses Superior_Set and Junior_Set of algorithm 6.2 in figure 6.12.

Var s.set, j.set: role set; $r, r_j, r_s$: roles;
BegiIf $Direct(target) \neq \bigcup(Direct(x_i))$
        Then abort                            /* Must keep privilege set invariant */
    Else begin
        1.  $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;                  /* Add new roles to system Roles */
        2.  s.set := Superior_Set(target);    /* Generate the superior set */
        3.  j.set := Junior_Set(target);      /* Generate the junior set */
        4.  $\forall x_i \in \{x_1, \cdots, x_n\}$ do             /* Assign privilege set to new role */
                $Direct(x_i) := rpset_i$;            /* Assign respective privilege sets */
        5.  $\mathcal{R} := \mathcal{R} - target$;               /* Delete target */
        6.  $\forall x_i \in \{x_1, \cdots, x_n\}$ do
            begi$\forall r_s \in s.set$ do add $x_i$ - $\rightarrow_s$;   /* Link New Roles to seniors */
                $\forall r_j \in s.set$ do add $r_j$ - $_i$;      /* Link New Roles to juniors */
            end;
        7.  $\forall r_i, r_j \in \mathcal{R}$ if $\Psi(r_i) = \Psi(r_j)$ then  /* Remove any duplicate roles */
            Begin
                $\forall r$ such that $r_i \rightarrow r$ do add the edge $r_j \rightarrow r$
                $\forall r$ such that $\rightarrow r_i$ do add the edge $r \rightarrow r_j$
                Delete all edges $r_i \rightarrow r$ and $r \rightarrow r_i$;
                Remove $r_i$;
            end;
        8.  $\forall r_i, r_j, r_k \in \mathcal{R}$ do if $\exists((r_i \rightarrow^{+} r_j \rightarrow^{+} r_k)$
            $\wedge(r_i \rightarrow r_k))$ then               /* Remove redundant edges */
                Delete the edge $r_i \rightarrow r_k$          /* Delete the direct edge */
    end;
end.                                          /* Horizontal_Partition */
```

Figure 6.15: Algorithm for Horizontal Partition

but do not alter the basic structure and relationships in the role graph structure. These may be addressed within the context of role-privilege authorization.

6.7.3 Correctness of the Graph Algorithms

In the foregoing sections we have the basic relationships and properties necessary for a role organization model. We proposed such a model to be a role graph and gave examples of algorithms for its manipulation. Note that several operations are possible on the role graph including, but not restricted to, those implemented by the algorithms presented in sections 6.7.1 and 6.7.2. This section emphasizes the need for correctness of operations on a given role graph. Operation correctness is based on an operation achieving its intended function while maintaining the properties of the role graph. An operation must not disturb the basic role graph structure hence role graph properties must hold both before and after the operation. Hence it is important to show that the operations implemented by the suggested algorithms are indeed correct, i.e. they achieve their function and preserve the role graph model properties enunciated in the foregoing sections. We shall offer arguments that demonstrate algorithm correctness of one of the algorithms. These arguments can then be extended to cover the rest of the given algorithms and any other operations that manipulate the role graph.

In any given application, correctness is based on some criteria defined within the system in which the application "executes". In our application, operation execution correctness if determined by the role graph properties. The operations given assume a role graph input in which all the properties and constraints hold. In our situation, the correctness criteria include:

1. achieving the intended purpose of the operation and, in case of failure to do so, aborting the operation and restoring the graph to its original state. The operation commits if and only if the operation's effects do not violate the properties of the role graph model. These graph properties ensure the two key characteristics of the role graph: *role graph proper form* (definition 6.18) and *privilege distribution optimality* (definition 6.19).

2. ensuring that operation execution leaves the role graph in a state in which all the properties and constrains pertaining to the role graph hold. Hence properties 6.1 $\cdots$ 6.7 and constraints 6.1 and 6.2 must hold.

3. ensuring that the operation causes no side effects other than those directly and indirectly possible via the operation. Side effects indirectly generated by an operation arise from any "normalizations" which derive from role graph properties. For example, if after an operation we find that for two roles r_i and r_j, $\Psi(r_i) \subseteq \Psi(r_j)$, then we must ensure that there exists a path $r_i \rightarrow^+ r_j$.

We choose the algorithm for role addition, i.e. algorithm 6.1 in figure 6.9, to demonstrate the correctness of our operations. We make the following conjecture:

Conjecture 6.1 *Given a role graph rg, a target role $target$ to be added to rg, a set of roles $j.target.set$ in rg to be the $target$'s juniors and another a set of roles $s.target.set$ in rg to be the $target$'s seniors, Role_Addition i.e. algorithm 6.1 of figure 6.9 preserves operation correctness.* □

Proof: The input graph rg must be such that all the role graph model properties hold. Our proof is based on the fact that once we assume that role graph properties hold for rg, then it is sufficient to show that no role graph property or constraint is violated by the operation to demonstrate operation correctness. Preservation of these properties ensures that role graph remains both in proper form (see definition 6.18) with privilege distribution optimality (definition 6.19). Our proof dwells on the three main issues listed above in this section. We proceed to show that, on the basis of these issues, Role_Addition, meets the correctness criteria.

1. Perhaps the most important characteristic of the role graph is its acyclicity. Where there is potential violation of acyclicity, the operation must abort. Role_Addition checks whether the intended junior ($j.target.set$) and senior ($s.target.set$) sets of the target to be added, as well as the target's privilege set, would cause cycles . We show that Role_Addition does not violate acyclicity.

   Since the graph is acyclic, we are concerned with the immediate neighbourhood of the target to be added. Now, assume that we end up with a graph with cycles.

   We know that cycles, i.e. $\exists r_i, r_j, r_k$ such that $r_i \rightarrow^+ r_j \rightarrow^+ r_k \rightarrow^+ r_i$, in the graph would arise in two ways: (a) due to the junior and senior sets, i.e. $\exists r_j \in j.target.set, r_s \in s.target.set, target$ such that it is possible to have $r_j \rightarrow^+ target \rightarrow^+ r_s \rightarrow^+ r_j$ or (b) through the privilege relationships and due to the *is-junior* and or *is-senior* relationships after the incorporation of the target into the role graph since we know that whenever we have $\Psi(r_i) \subseteq \Psi(r_j)$ we must have $r_i \rightarrow^* r_j$.

To deal with (a), Role_Addition checks, as the first step that the junior and senior sets would not cause such a cycle. If this is potentially possible, the addition is aborted. The operation proceeds only when there is no potential cycle between the junior and senior sets. Therefore, Role_Addition introduces no cycles on insertion of the target role *target* with *j.target.set* and *.target.set* as its junior and senior role sets, respectively.

With respect to (b) Role_Addition checks in step 3 to ensure that no such potential cycles occur since, due to privilege relationships, should there be a potential cycles on insertion of *target*, Role_Addition aborts.

Conclusion: Role_Addition introduces no cycles into the role graph *rg*.

2. We make the claim that Role_Addition does indeed leave the role graph in a state where properties 6.1 $\cdots$ 6.7 and constraints 6.1 and 6.2 hold.

   We note that Role_Addition causes no redefinition of the role graph properties. Therefore, to show operation correctness with respect to role graph properties, it is sufficient to show that the operation violates none of the properties. We proceed to demonstrate this on property by property basis.

   The algorithm does not affect privilege monotonicity, i.e. property 6.1. It preserves privilege uniqueness, i.e. property 6.2 by ensuring that no target role is added with identical privilege set as an existing role (see line 2). Where this is potentially possible, the algorithm treats this role as the target role and adds necessary arcs to the juniors and seniors of the target. Role_Addition touches neither MaxRole nor MinRole, i.e. properties 6.3 and 6.4, respectively. Moreover, each role in the graph still remains a role graph as per property 6.5. It does obey the privilege set invariant property 6.6 since Role_Addition execution must be authorized, like any other operations that execute in the system. Therefore all the privilege set changes it causes must be authorized as well. Further, the mandatory path property 6.7 holds as the algorithm, in steps 3(d) and 3(e), makes sure that all paths are present.

   With respect to constraints, Role_Addition maintains acyclicity (constraint 6.1) and hence it doesn't violate this constraint. Moreover, the path privilege monotonicity (constraint 6.2) still holds as Role_Addition utilizes this to reorganize the roles in the system.

Conclusion: After the execution of Role_Addition properties 6.1 $\cdots$ 6.7 and constraints 6.1 and 6.2 hold.

3. Finally, we claim that Role_Addition introduces no side-effects other than those directly and indirectly associated with its execution.

   Assume, that it did introduce such side-effects. This means that certain changes would occur within the role graph by other means, other than that those inferable via the properties in the system. However, as we have shown, Role_Addition causes only changes that derive from the characteristics of the role graph. Hence all changes must thus derive from Role_Addition directly (e.g. by the addition of the target role and its privileges) and indirectly (e.g. via the application of role graph properties to ensure inferred relationships hold as per the said properties.)

   Conclusion: Role_Addition causes no changes other than those directly and indirectly required by the operation and the role graph properties.

Main Conclusion: We conclude that Role_Addition does indeed execute correctly by preserving the properties of the role graph, meeting role graph constraints and introducing no other side effects other than those directly or indirectly related to the operation. It leaves the role graph both in proper form and with optimal privilege distribution. Thus we have established that conjecture 6.1 holds. □

To show that the rest of the suggested algorithms execute correctly, we could use similar arguments.

6.8 Comparison with Hierarchies, Privilege Graphs & Others

The model presented here has the expressive power of other role organization structures. In this section we demonstrate such power in the simulation of hierarchies [TDH92] and privilege graphs [Bal90].

The role graph model can simulate a hierarchical organization. We can convert a role graph into a tree (hierarchy) and vice versa. To obtain a tree from a given role graph, we designate **MaxRole** as the root of the hierarchy and do a recursive *bread-first* or *depth-first* traversal for every node with a relationship with **MaxRole**. A given path terminates when **MinRole** is encountered which forms the leaves of all paths in the resulting tree (hierarchy). The resulting hierarchy has "duplicated" leaves which in each case in the **MinRole**. In general, there is duplication whenever the greatest lower bound of two or more roles is a role different from **MinRole**. We have a duplication of just the leaf nodes only when **MinRole** is the only common lower bound of any two roles. This tree contains *all* paths present in the associated role graph. In going from a tree to a role graph, we designate the root of the tree to be **MaxRole**, do a depth-first traversal of the tree and equate nodes whenever equal privileges are encountered. The resulting role graph can then be augmented with **MinRole** if necessary. The advantage with the role graph is its *compactness*, i.e. shared nodes lower in the hierarchy, need not be duplicated. This is a major advantage in that it reduces the extent to which shared privileges are scattered among roles which makes the task of tracking their use easier.

To simulate privilege graphs [Bal90], attach to every role an associated functionality that specifies the associated duty requirements/title/etc. With the role's access control list (racl) acting as the user/group node (figure 6.2), it is possible to determine the authorized users for any role. An authorized user's access rights are determined by the effective privilege set $\Psi(r)$ of the associated role r to which the user is authorized. Further, remove **MaxRole** and assign its direct privileges to roles with a direct partial privilege relationship with it. The result is a privilege graph.

To simulate lattices, we let the greatest lower bound set (glbs) and the least upper bound set (lubs) represent a lattice's greatest lower bound (glb) and least upper bound (lub), respectively.

Finally, although this model is based on subsets with an acyclic graph, it is different from the Bell and LaPadula Model (BLPM). Moreover, although both are meant for security application, they have different approaches to realizing protection. The BLPM relies on subsets, acyclicity and is static. However, it is based on the classification of information as opposed to the execution of operations as is the case in our model. The BLPM specifies two simple operations of either read or write access depending on object classification and subject clearance. This approach realizes multilevel security. In our model, privileges represent pre-defined executions designed in a manner intended to realize certain desired functionality in a system. These operations are designed from considerations of desired system functionality. Once defined, the operations are distributed among roles in the system in the manner that suits organizational requirements. The executions can be simple reads and writes. They can be a combination of simple reads and writes. But they can also be complex executions of such methods in object-oriented programming. These operations need not merely alter or return the information relating to a given object but can also create other objects and invoke other operations.

In the BLPM, once classification has been done, access to information is governed by the simple security property and the *-property. Its specification is static. In our model, execution of privileges can cause the assignment or revocation of privileges pertaining to some role. In that respect, our model is dynamic.

6.9 Summary

This chapter has presented a basis of role organization and proposed a framework for such organization. We discussed common role organizing structures to enable us "extract" properties for the formulation of role organization. The two key properties are the *acyclic* nature of the role organizing structures and the *monotonic* privilege relationship among roles in a given path in the organizing structure.

We discussed basic role relationships (partial, common and augmented) based on privilege sharing between roles. Starting with these two role relationships, we generalized to cases of more than two roles. Through such extension, we proposed a role graph framework that preserves the key properties and offers differentiated privilege sharing. The concepts of maximum and minimum privileges were introduced for purposes of easing the modeling. With these two it is possible to completely specify the semantics of the operators introduced for the various role relationships.

Acyclicity of the resulting structure and *monotonicity* of role privileges in a given path ease the task of administration and analysis of privilege distribution in a system; privileges need only be defined at the most specific location in the structure.

We formulated a means of specifying role independence and role coupling to help study the extent of privilege sharing. Role independence (revisited in the next chapter) allows us to assign non-conflicting assignments without worrying about shared responsibility; independent roles have no shared privileges/responsibilities/etc[8].

Role management strategies were also presented. Using the graph structure derived, we present a role administration scheme as well as associated algorithms. Thus we discussed strategies for role addition, deletion and partition.

To illustrate the expressive power of our role graph structure, we presented a means

[8]As will be seen in the next chapter, tighly coupled graphs are unlikely to be useful where partitions are required to forestall conflict of interest.

for simulating hierarchies [TDH92] and privilege graphs [Bal90]. This was intended to demonstrate that our model is at least as expressive as the two structures. Similar simulations can be done for lattices [RWK88, RWBK91] (the closest structure to our model) and Ntrees [San88, San89] (which have similar properties with lattices).

The key **contribution** of this chapter is the exploration of the basic relationships between roles which are then applied to the modeling of role organization. We also addressed the key properties necessary for differentiated privileges for different roles in such a role organization. We found that we need basic role relationships (partial, common and augmented privilege sharing). The key properties were also found to be acyclicity of role organization structures and the monotonic privilege relationship for roles in a given path in the structure. We formally defined a *pseudo* lattice structure, also referred to as a role graph, that incorporates these properties. We presented a means of role administration for roles in the structure that preserves the graph properties. In particular we formulated methodologies (in the form of algorithms) for role addition, deletion and partition.

We defined the concepts of role independence and role coupling. Coupling refers to the extent of privilege sharing among roles. Independent roles have no coupling between them. The coupling factor introduced can act as an indication of the extent of privilege sharing in the role graph.

CHAPTER 7

REALIZING COMMERCIAL INTEGRITY

7.1 Introduction

In the previous chapter we presented a role graph model for role organization. This chapter explores an application of the model, along with the object and transaction models of chapters 3 and 4, respectively, to realize the principles of the Clark and Wilson model [CW87]. The object model provides a framework for modeling objects and manipulating them. The transaction model forms the basis for transaction execution of processes in our model. Indeed, all our executions are either transactional in nature or execute within some transaction. Finally, the role graph of the previous chapter provides the basis for role relationships and hence role organization.

The principle requirements of the Clark and Wilson model [CW87] have been summarized in chapter 2. Among them are requirements such as database items (those requiring integrity) being constrained data items (CDIs), that only certified transformation procedures (TPs) manipulate the CDIs, that the TPs are certified to take CDIs from one correct state to another, that there are verification procedures (IVPs) that assure the integrity of the CDIs, etc. In general, all data transformations are required to be designed as well-formed transactions (WFTs) where a WFT is a program that has been certified to maintain the integrity of the data it manipulates [CW87, Kar88, Lee88, San91, Tho91]. The manner in which O-O executions meet transactional requirements has been given in chapter 4; method execution must be either transactions or be part of transactional executions. In this chapter, we shall utilize the transaction framework to meet the integrity requirements of the commercial integrity model [CW87]. We shall demonstrate the well-formedness of the transaction procedures executed in our model. From the point of view of the O-O paradigm, these transactional requirements make the object interface transactional. Further, objects (at least those which require integrity to be maintained) will be handled as CDIs as required by the commercial security model.

Integrity is concerned with modification of information. Integrity policies can be seen as falling into two broad classes: whether the modification is authorized and whether the modification results in a consistent state [SD87]. Consistency encompasses integrity rules, recovery management and concurrency control. The following sections will address realization of the Clark & Wilson model properties within this perspective.

Separation of duty is another major requirement for commercial database integrity [CW87, Kar88, Lee88, San91, Tho91]. The Clark and Wilson model [CW87] proposes separation of duty as a means of dispersing the exercise of responsibility in commercial environments. Using *object histories* introduced in chapter 3 and role authorization, we shall demonstrate how our formulation realizes separation of duty. Object histories facilitate the tracking of users that have participated in the processing of some object along with the events they executed in the process. The authorization specifies the user-TP-CDI relationships. Further, with the role graph organization of chapter 6 we associate subtasks in execution sequences to roles in paths in the role graph. This is equivalent to having some execution sequence mapped onto a path in the graph. By keeping the history of execution from one step to another along the path, we enforce what we term *in-path* separation of duty.

Another important principle in commercial security is *conflict of interest.* Like separation of duty, it is intended to disperse the exercise of authority. However, whereas separation of duty involves subtasks of one task, conflict of interest connotes totally different tasks. There exists conflict of interest where there is violation of integrity requirements in which some user executes two tasks that conflict. This chapter will address this too and uses the role graph of chapter 6 for the purpose. We shall show that independent role groups in a role graph in which a user is authorized to roles belonging to one such group, forestalls conflict of interest.

The rest of this chapter is organized as follows: In section 7.2 we discuss the role concept further, especially in reference to the object model of chapter 3. In particular, we note that privileges, since they can be complex, can be defined as object-method pairs. Each such privilege acts as a *window* to object information. Associating different methods with different roles distributes the object interface across different roles with each role having its own set of windows into object information. It is up to system designers to determine the manner of distribution of object interfaces to meet system processing requirements. Roles themselves can be seen as windows to database information. Such windows can be expanded and/or narrowed depending on the associated list of privileges; adding a privilege to a role's privilege list expands this window while revoking a privilege from the list narrows the associated window.

Section 7.3 relates roles to the transaction model of chapter 4. In this respect, method executions (as defined in section 7.2) must be either transactional or execute within transactional executions. The main aim is to meet the transactional requirements of WFTs. The transaction model of chapter 4 offers the framework for these executions. As well, we demonstrate transaction well-formedness of the transactional executions associated with CDIs.

In section 7.4 we address the issue of authorizations to roles in which we discuss both *user-role* and *role-role* authorizations. The former gives an authorized user/group access to the privileges defined in the role. The latter gives a role authorization to privileges of another role. In essence, this serves to define role relationships given that a role authorized access to another role's privileges has a *partial* privilege relationship (see chapter 6) with the other. Indeed, as will be demonstrated, the whole idea of role organization can be captured using role-role authorizations.

Further discussion in this section addresses the issue of authorizations and the role graph. A user authorization to a role gives such a user access to both the direct and indirect (those defined in its junior roles) privileges of the role. If the effective privilege authorization can be used as a measure of a user's authority, then the authorization scheme can be seen as conferring authority to users while the role organization structure can be seen as some authority structure. For instance, for any given path in the role graph, a user's authority will be determined by the position (in the path) of the role to which the user is authorized.

Separation of duty is the subject of section 7.5. We demonstrate how, from our formulation, we meet one of the enforcement requirements in the Clark and Wilson model. This is the requirement that system security officers (SSOs) associate users with transformation procedures and the objects that the procedures operate on. In our model, since all users execute privileges in authorized roles which are defined as methods (the transformation procedures), we show that we can "compute" for each user an *authorization scope* that "lists" a user's privileges. The use of audit information to determine whether to grant access is another important requirement of separation of duty. Our solution is to use object histories to keep track of an object's audit trail information and hence make it readily available for use. Object histories are part of audit trail information hence keeping this information within the object itself, makes it readily accessible. As Karger [Kar88] notes, scanning the audit trail for the same information would be a major processing task.

A path in the role graph can be associated with a given task. Such a task's subtasks would then be associated with roles in the path which effectively maps the subtask processing order to a path in the role graph. Imposing separation of duty on the task processing amounts to what we term ***in-path*** separation of duty. We start with the case of one user authorized to one role and extend it to group authorization to the role.

Conflict of interest, also a major issue of concern in commercial security, is discussed in section 7.6. In particular we present a means for managing conflict of interest within the framework of the role organization.

Section 7.7 is a review of our formulation with illustrations of how it meets the requirements of the Clark and Wilson model. Going through the suggested properties in the Clark and Wilson model, we demonstrate how our formulation meets respective properties.

Section 7.8 presents the summary, conclusions and the key contributions of this chapter.

7.2 Roles and the Object Model

This section discusses the role concept in the context of object-oriented (O-O) principles. The basic idea is that since methods form the ***only*** means of access to object information, they form the object interface. Further, since each method gives access to object information in a particular form, a method can be seen as a ***window*** to object information. Moreover, methods, being the only access modalities in the O-O paradigm, can form the basis for privilege definition. Consequently, the least unit of privilege definition is an object-method pair. Hence, each such privilege acts as a ***window*** into the object.

Given that methods constitute the object interface, associating different methods with different roles ***distributes*** the object interface among the roles with each role having its own (set of) window(s) into object information. Such windows can be expanded and/or narrowed by varying the entries of the roles' privilege lists. It is up to a system designer to determine the manner of distribution of an object interface to meet system requirements. From this basis we modify definitions for privileges and roles to reflect O-O contexts.

7.2.1 Roles, Privileges & O-O Interfaces

Roles offer differentiated access to database information based on their associated privileges. In the O-O paradigm, information is held in the state of objects and is accessible via methods. In this section we combine the two concepts to exploit the advantages of role-based protection and those of the O-O paradigm.

In the O-O paradigm, each object is an instance of some type/class defined in the system. Object information access takes place only via the methods defined in the type/class. The O-O paradigm is very suitable (almost natural) for this scheme where objects are assigned types/classes with methods in the type/class being the well-formed transactions. Essentially, the designer of an application specifies the types/classes in the database by defining the attributes (and their types/classes) of the instances and the methods that are applicable to them. Objects defined in the database must then be instances of the defined types/classes in the database. Their behaviour is determined by the methods that operate on them.

An important issue is the assignment of privileges to roles in a system. This can be seen as the composition of roles which determines what a user authorized to the role can accomplish while executing privileges in the role. In other words, what information is accessible via the role? What form of access is it? Is it information update or just access without modification of information? The system designer would assign, to the associated role, methods that perform the desired function.

Role privilege assignments can be seen as role-resource/object authorization. In an O-O environment, this can be realized via role "authorization" to the associated method, i.e. the inclusion of a method in the role's privilege list ensures that the object can be accessed via the role in the manner determined by the nature of the method execution. From definition 3.1i in chapter 3 we know that each method can be seen as a partition of the object interface. i.e. given an object, o its interface $OI(o)$ is the set of methods $Methods(o)$ associated with the object, i.e. $OI(o) = Methods(o)$. A partition of the interface (part(OI(o))) is a subset of the object interface OI(o), i.e. $part(OI(o)) \subseteq OI(o)$ or $part(OI(o)) \in 2^{OI(o)}$ (see definition 3.10). In an O-O environment, such partitions are central to role definition and hence authorization. Consequently, we redefine privilege. and to avoid confusion, we term this an *o-privilege*:

Definition 7.1 o-privilege: *An o-privilege is a pair (x,part(OI(x)) where x determines a unique object (resource) and part(OI(x)) is some partition of the interface OI(x) of x.* □

As seen elsewhere, x can be any object, resource, etc. that is accessed via some interface $OI(x)$.

The difference between definitions 5.2 and 7.1 is that, in the former, access modalities can be any types such as read, write, execute. In the latter, the access modality must be a partition of the O-O object interface. In other words, it must be, at least, a *legally* defined access method for the object. This implies that the least unit that can constitute a privilege is a object-method pair, which leads to the following lemma.

Lemma 7.1 *In an O-O environment, the least unit of privilege definition is an object-method pair.* □

Proof: Consider the definition 5.2. x refers to an object while m is a non-empty set of valid access modalities for x. The least unit of privilege definition must then be the pair (x, m) with m containing at least one valid access modality of x. In an O-O oriented environment valid access modalities are methods.

Suppose that in such an O-O environment some access modality is not one of the methods defined for the given object. This would suggest that there exists some valid access modalities for the object other than the methods associated with x. However, we know that in an O-O environment (like that defined in chapter 3) only methods are valid modalities of access to objects. Consequently, it follows that the least unit of access must be some method in the object's method list. The least unit of privilege definition would be the object x itself and such a least unit of access. Therefore in an O-O environment, the least unit of privilege definition *must* be some object-method pair. given object. □

The method itself could be complex (see the method invocation tree in figure 3.2) but what is important is that for the object, such a method must be defined within the types/class of the object. Moreover, as we shall see in the next section, such access modalities can indeed be transactional.

For the rest of this thesis we do not make a distinction between definitions 5.2 and 7.1.

Example 7.1 **Consider the cheque issuing process of chapter 4 in which two "signatures", of a clerk and supervisor, are required to be appended onto a cheque, and where the clerk's signature must come before that of the supervisor. The cheque object is an**

instance of a type CHEQUE with two methods, clerk and supervisor, which append (update) the clerk and supervisor signatures to the cheque object, respectively. We use the type definition of example 3.1 on page 39.

Let method clerk be implemented to update the PAYEE, PAYEE_ID, AMOUNT and SIGN_1 attributes with the payee name, payee identifier, the amount of the cheque and signature, respectively. Then the audit trail is updated with the appropriate information and the cheque is "dispatched". On "receipt", the supervisor invokes method supervisor which, among other things, updates SIGN_2, audit trail before dispatching the cheque for, say, payment. A security system will specify authorization to the appropriate methods such that subjects assigned to the clerk and supervisor roles can execute the clerk and supervisor methods, respectively.

□

7.2.2 O-O Roles & Object Information Windowing

In determining and enforcing access control, the windowing effect discussed in the previous section facilitates differentiated access to object information. This can be achieved via authorizing different users to access different portions of object information via the associated interface partitions. By explicitly (or implicitly) basing authorization on these partitions of the interface (windows) to different roles, we can effectively enforce differentiated access to object information to different users.

Role definition in an O-O environment must recognize and take advantage of this fact. The definition must be in terms of the o-privileges. Hence a role can be seen as grouping different o-privileges (objects and their interface partitions). A role definition scheme $\mathcal{RDS}$ defines roles $\mathcal{R}$ from given objects $\mathcal{O}$, associated methods $\mathcal{M}$ and role names $\mathcal{RN}$, i.e.

$$\mathcal{RDS} : \mathcal{O} \times \mathcal{M} \times \mathcal{RN} \mapsto \mathcal{R}$$

It is an assignment of partitions of the interfaces to roles. This can be specified as:

$$\mathcal{RDS} : \mathcal{O} \times \mathcal{M} \times \mathcal{RN} \mapsto \mathcal{R} \subseteq \mathcal{RN} \times \mathcal{O} \times 2^{OI(\mathcal{O})}$$

We refer to roles defined in terms of o-privileges as o-roles. Formally:

Definition 7.2 *<u>o-role:</u> An o-role is a named collection of o-privileges; i.e. o-role = (o-rname, $\{\cdots, int_{i,j}, \cdots\}$) where each of the $int_{i,j}$ is of the form $part_i(OI(o_j))$ and each object o_i can be accessed via the respective partition in $\{\cdots, int_{i,j}, \cdots\}$ through the o-role.* □

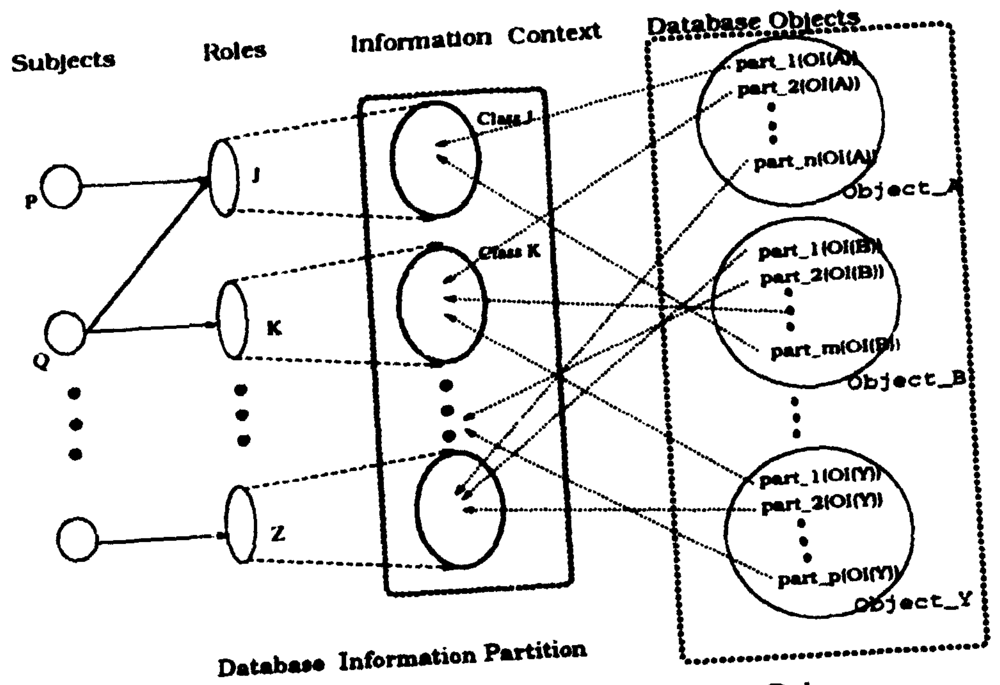

Figure 7.1: Object Interface Distribution Over Roles

The differences between definitions 5.3 and 7.2 follow from the definitions of privilege and o-privilege.

A key advantage with this approach is that in designing interface partitions and specifying access control, one can enforce the principle of *least privilege* by ensuring that only necessary partition(s), that avail as much information as required to a user, is(are) authorized to a role. Moreover, this introduces further flexibility to our role window management whose variation can be effected either by privilege revocation/addition or via narrowing/widening of the interface associated with given privileges in the role's privilege list.

A direct outcome of o-privilege definition is that interfaces associated with different privileges can overlap. We say two privileges are related if they have intersections in their interfaces. In other words, the two privileges pertain to the same object and their access modes overlap. Such overlaps can result in overlapping contexts (see section 5.5.4). Formally:

Definition 7.3 Related Privileges: *Two privileges pv_i and pv_j are related if they refer to the same object and their interfaces overlap, i.e.* □
$(pv_i.x = pv_j.x) \wedge (pv_i.part(OI(x)) \cap pv_j.part(OI(x)) \neq \emptyset)$

Definition 7.4 Role Relationship: *Two roles r_i and r_j are related if they have shared privileges and/or if any of their privileges are related.* □

The O-O paradigm is very suitable (almost natural) for this scheme of role-based protection in which we have roles authorized to execute specific partitions of some objects (see figure 7.1). The resulting effect is the distribution of an object's interface across roles. Role privilege assignment is the task of the system security function. It can be defined in terms of privileges and its enforcement during execution is a mandatory system function. Neither the users nor their processes can carry out the tasks of a function and the administrative aspects of the privileges that accomplish the task. In other words, a user cannot alter the system security function even when this is defined in terms of privileges.

7.3 Roles & The Transaction Model

In section 7.2 we outlined role definitions in terms of object interfaces in the O-O paradigm. We made the observation that the smallest unit that could possibly specify such a privilege is an object-method pair. In this section we take the matter further to define privileges in terms of transactions with the transaction model of chapter 4 being the framework for our transaction executions. Another important concern in this section is the *well-formedness* of the transactions.

The formulation in this section pertains to the correctness of the transformation procedures and the correctness of the state of the objects they manipulate. Object state correctness, and hence its validity, depends on its associated integrity constraints, a recovery management scheme (e.g. where a transformation must abort) and concurrency control property to ensure isolation [SD87].

7.3.1 Roles & Transaction Executions

A transactional execution (TE) (also from chapters 3 and 4) is one bounded by a **BEGIN** and either a **COMMIT** or **ABORT** with the executions bounded by these being subject to *isolation*. The execution's effects after a **COMMIT** take permanent effect. Such execution's effects will be undone or compensated for in the case of an **ABORT**. Further, the whole execution will take the database from one correct state to another. This is a requirement of the ACID properties (see chapter 4) in transaction models. A TE (see page 46 in chapter 3) is of the form:

```
pre-conditions
IF preconditions then
        BEGIN
                { Execute operations ensuring isolation }
                IF commit conditions then
                        COMMIT { operations }
                ELSE ABORT { operations }
        END.
```

Given that the nature of method execution in an O-O paradigm is a matter for the system designer, we can define such executions either as *transactions* or as *transactional executions*. In the former, a method is itself a transaction while in the latter, the method executes as part of some transaction. We say the latter method is *transactional*. Hence like the object interface (OI) of the previous section, we can define a transactional object interface (TOI) for objects in the database as being an interface in which every interface partition is a transaction or is transactional. An object which is accessible only via transactions or transactional executions is said to have a *transactional interface*. An object that is partially accessible via a transactions has a *partial transactional* interface and one that is not transactionally accessible is said to have a *non-transactional* interface. Formally a transactional interface is defined as:

Definition 7.5 *Transactional Object Interface (TOI): An object interface in which all the methods are either transactions or are only in transactional executions.* □

Definition 7.5 implies every method associated with such an interface has transaction properties. Consequently, the object itself may be viewed as "transactional" in that all operations on it are either transactions or transactional executions. Thus there will be no operation that will "mess" up the object with such an interface.

From definition 7.5 we define a transactional privilege (to-privilege) and a transactional role (to-role).

Definition 7.6 *to-privilege: A to-privilege is a pair (x,part(TOI(x)) where x refers to an object (resource) and part(TOI(x)) is some partition of the transactional interface of x.* □

Definition 7.7 *to-role: A to-role is a named collection of to-privileges, i.e.* *to-role* $=$ *(to-rname,* $\{\cdots, toint_{i,j}, \cdots\}$*) where each of the* $toint_{i,j}$ *is of the form*

$part_i(TOI(o_j))$ and an object o_i can be accessed via the respective transactional partition $\{\cdots, toint_{i,j}, \cdots\}$ through the to-role. □

For the rest of this chapter we shall be referring to definitions 7.6 and 7.7 when we talk about privileges and roles, respectively.

Our formulation is intended to meet the constrained data item (CDI) requirements of the Clark and Wilson model in which all CDI accesses must be WFTs. We impose the following constraint:

Constraint 7.1 Constrained Data Item Access Constraint: *All CDIs must have only transactional interfaces.* □

Constraint 7.1 implies that all CDI classes and their subclasses must have transactional interfaces. Every CDI's interface and all its partitions must be transactional.[1] Observe the effect of this imposition. It ensures that the database, if it meets the *pre-conditions* of an execution, will be left in a correct state after the execution. The pre-conditions ensure that both the database and the TP are in a correct state. Pre-conditions designate the fact that when they are met,they guarantee the associated post conditions. This way the TP is assured to execute correctly and hence leave the database in a correct state.

In our formulations, the post conditions are the transaction's *commit conditions*. These ensure that only correct executions will have durable effect on the database. Transactions are aborted when these conditions are not met. Effects of aborted executions are undone or compensated for accordingly. Both undo and execution compensation are assured to leave the database in a correct state.

An important observation to make is that since transactional executions are based on the O-O method invocation scheme, they can be complex and can be nested to an arbitrary level. Each transaction execution in our formulation assumes such a complex environment in which every such execution is a node in a transaction execution tree (see figure 4.1).

7.3.2 Transaction Execution Well-Formedness

Section 7.3.1 outlined role definitions in terms of transaction executions. This section demonstrates the well-formedness of these executions. Well-formedness is a key requirement of the Clark and Wilson model, hence showing the well-formedness of executions is a step towards satisfying the requirements of the model.

[1] Indeed, this is the case from chapter 3 and is made here for the purpose of emphasis.

The concept of well-formed transactions requires that users manipulate data only in prescribed ways; users cannot manipulate data in an arbitrarily manner [CW87]. Further, every manipulation must not only be prescribed, but also its effects on the data it manipulates must be recorded in some audit log. Audit information from the audit log is useful in the reconstruction of the actions on the data, i.e. the history of the executions on the data. The nature of the audit log entries pertaining to particular types of objects would depend on the nature of the information necessary to "detect" any "mistakes" in the manipulation of the object. Such information would also be used to trace the users responsible for the execution. For instance double entry book-keeping ensures that a balance is kept all the time. The kind of information kept about objects of this nature would be related to the credits and debits that form the basis for determining a balance.

In our model, each object is an instance of some class. Hence it can only be manipulated by the methods defined in the class. The class prescribes the nature of the manipulation on its instances. Secondly, for instances of the CDI class and its subclasses, these executions on objects have transactional properties. Thus they have permanent atomic effects on the objects. Thirdly, not only does each object track its history, but also the system's audit log can be reconstructed from the individual histories. However, where it is not considered a major expense, such a central log can be maintained alongside object histories. This will require that every update of object history will be entered in the system log as well (see further discussion in section 7.5.2). Finally, since all the executions on CDIs are transactional, their effects are assured to be atomic, consistent, isolatable and durable past the life of the execution.

Our transactions execute within the transaction framework of chapter 4 which include the well-formedness of the executions as outlined in section 2.4.1. An important recognition is that at *database initialization*, all code must be verified to execute correctly within the requirements of the *pre* and *post* conditions. Further, any newly introduced code is verified only with respect to existing code with which it interacts. This negates the need to verify the system all over again. The key idea is that to the extent that a given execution does not interact with other pieces of code, it does not affect it. Hence if such code were verified at initialization, it need not be verified all over again. Since all changes on data objects and code are retained in the system's audit trail, any malicious change would not go undetected.

The pre and post conditions that govern execution ensure that with correct input (as specified in the pre conditions) there would be correct output (as specified in the post conditions). Otherwise, failure to meet any of these means that the execution will be incorrect and must thus be aborted. Once both the pre and post conditions are met, the execution can be regarded as correct.

7.4 Users, Roles & Authorizations

An important aspect of information integrity is user authorization [SD87]. We regard user authorization to be mandatory given that no user will access information without authorization, and that no one user can modify security information pertaining to oneself. Information access is realized via invocation of transformations. Therefore, user authorization to information is realized via authorization to system transformation procedures.

This section addresses the issue of authorizations. In particular we shall concentrate on ***user-role*** and ***role-role*** authorizations. We discuss role-role authorizations and how they capture role relationships.

7.4.1 User-Role Authorizations

User-role authorization is important in system security administration as it determines the users' access rights. A role with a defined user-role authorization, in the form of a role access control list, is termed secure (see definitions 5.12 and 5.13). In a secure system, it is required that all roles be secure roles. In other words if we have a universal set of roles $\mathcal{R}$ and a universal set of secure roles $\mathcal{SCR}$, the two sets must be equivalent for a secure system. This is captured by the following constraint:

Constraint 7.2 Secure Role Constraint: *In a secure system, every role must be secure.* □

The effect of user role authorization is to generate relations of the form: $(rname, \{(o_i, m_j), \cdots\}, \{\cdots, id_p, \cdots\})$ where $rname$ is the role name, o_i the object, m_j is a method and $\{\cdots, id_p, \cdots\}$ is the access control list.

From a user's point of view, user-role authorization determines the user's access rights in a system. The cumulative access rights of such a user in a system, termed a ***user's access scope***, determines what a user can access in the system. It determines the nature of information at the reach of the user and the nature of access (read, update, etc.). The scope is determined by the union of all privileges authorized to the user.

To compute a user role authorization scope, we must generate all roles to which the user is authorized. These are termed the user's role scope. Formally:

Definition 7.8 *User Role Scope ($\mathcal{URS}$): A user's Role Scope (URS) is the set of all roles that the user is authorized to. Given a user identifier and the set of secure roles in a system, a user role scope function enumerates all roles to which the user is authorized.* □

Given an authorization scheme and some $id \in ID$ we can generate a user's *authorization scope* of the form: $(id, (rname_a, rname_b, \cdots))$. To generate this list of roles we enumerate all those roles to which the user is authorized. Let $\mathcal{URS}$ be the function that enumerates the roles associated with a given user i.e. $\mathcal{URS} : \mathcal{ID} \times \mathcal{SCR} \mapsto 2^{\mathcal{SCR}}$.

$$\mathcal{URS}(id, \mathcal{SCR}) = \bigcup_{(\forall r \in \mathcal{SCR}) \wedge (id \in r.racl)} (r)$$

The condition $id \in r.racl$ "filters" out the roles to which the user is authorized.

A user privilege authorization scope is the set of all privileges to which the user is authorized. It is an enumeration of all privileges in the roles to which the user is authorized. Formally:

Definition 7.9 *User Privilege Authorization Scope ($\mathcal{UPS}$): A user's privilege authorization scope is the set of all privileges accessible to the user via the roles authorized to the user.* □

Having generated the user role authorization scope, we can enumerate their privileges to generate, for a particular user, a relation of the form $(id, (m_j, (o_1, o_2, \cdots)), \cdots)$.

Let $\mathcal{UPS}$ be a user privilege authorization scope function. It can be seen as a mapping
$\mathcal{UPS} : \mathcal{ID} \times \mathcal{SCR} \mapsto \mathcal{PV}$. It enumerates the privileges associated with some user. Hence given $id \in \mathcal{ID}$ and some $r \in \mathcal{SCR}$, we have

$$\mathcal{UPS}(id, \mathcal{SCR}) = \bigcup_{(\forall r \in \mathcal{SCR}) \wedge (id \in r.racl)} \Psi(r)$$

Given a system with users $\mathcal{ID}$, secure roles $\mathcal{SCR}$, objects $\mathcal{O}$ and methods $\mathcal{M}$, an access strategy Φ can be defined as:

Definition 7.10 *Access Strategy: An access strategy Φ is of the form:*
$\Phi : \mathcal{ID} \times \mathcal{SCR} \times \mathcal{O} \times \mathcal{M} \mapsto \{true, false\}$. □

If Φ_1 is an access strategy of the form specified in definition 7.10 involving $id \in \mathcal{ID}, r \in \mathcal{SCR}, o \in \mathcal{O}$ and $m \in \mathcal{M}$ we have:

$$\Phi_1(id, r, o, m) = \begin{cases} \text{true} \iff & \text{id} \in \text{r.racl} \\ & \wedge\ (\text{o, m}) \in \text{r.pset} \\ \text{false} & \text{Otherwise} \end{cases}$$

The condition id $\in$ r.racl (user-role authorization) ensures that the current user is authorized to execute the role while (o, m) $\in$ r.pset ensures that there is an associated privilege defined in the role; (o, m) can be defined as any subset of the authorized interface in the role.

Example 7.2 Consider the cheque process of example 7.1 and let the cheque object have the same methods clerk and supervisor. To associate these methods with roles, define two roles, CLRK and SPV, corresponding to clerks and supervisors, respectively. CLRK and SPV are then authorised to execute methods clerk and supervisor, respectively. This leads to role definitions of the form: (SPV,{(cheque,supervisor)}) and (CLRK, {(cheque,clerk)}) which effectiyely distributes the cheque object interface to two roles.

Next, individuals are authorized to execute the roles. For instance (say) John and Margaret are authorised for the CLRK and SPV, respectively. The roles with their access control lists now look like: (SPV,{(cheque,supervisor)}, {Margaret}) and (CLRK, {(cheque,clerk)},{John}). □

7.4.2 Roles, Explicit & Implicit Privileges

Roles are assigned privileges which determine what a user (or user processes) can do in executing in that role within the system. Privileges are determined *explicitly* (i.e. directly, e.g. by assignment) or *implicitly* (i.e. indirectly, e.g. by inference). Explicit role privileges are defined with the role specification while implicit role privileges are computed from the junior list of the role. The effective role privileges of a given role is the union of its direct and indirect role privileges.

In chapter 6 we saw that from role relationships we can have *partial* privilege relationships between roles. For instance if we have A→B it means $\Psi(A) \subseteq \Psi(B)$. In this case A's privileges are implicitly available to B. Moreover, the effective privileges of B are the union of B's direct privileges and those of A, i.e. $\Psi(B) = Direct(B) \cup \Psi(A)$. This privilege relationship can be seen as the case of authorization of B to A's privileges.

Consequently, if we have some path in our role graph (see chapter 6) of the form A→B→C, we have a privilege relationship of the form: $\Psi(A) \subseteq \Psi(B) \subseteq \Psi(C)$. In this case both A's and B's privileges are implicitly available to C.

Example 7.3 **Let $Direct(\Psi(A)) = \{1\}, Direct(\Psi(B)) = \{2,3\}$ and $Direct(\Psi(C)) = \{4,5,6\}$. Moreover, it is also given that A→B→C. Then $\Psi(A) = \{1\}, \Psi(B) = \{1,2,3\}$ and $\Psi(C) = \{1,2,3,4,5,6\}$.** □

Let $\preceq$ denote the precedence relationship between two roles in a given path $\cdots \rightarrow r_i \rightarrow r_j \rightarrow \cdots$, and $r_i \prec r_j$ when there is a path of the form:

$$\cdots \rightarrow r_i \rightarrow \cdots \rightarrow r_j \rightarrow \cdots$$

For a given role r in the path, we have:

$$\Psi(r) = \bigcup_{r_i \preceq r} \Psi(r_i)$$

7.5 Roles & Separation of Duty

A major property of the Clark and Wilson model is the requirement for the enforcement of separation of duty, a principle that requires that users that have been involved in the procedure are barred from further participation in the execution of the same procedure. Thus a clerk that has been involved in the draft of a voucher cannot approve the same voucher in the supervisor role. The same principle would bar a manager that acted in the position of a clerk from approving the same voucher. Separation of duty is meant to limit conflict of interest and guard against fraud by dispersing authority among different individuals. For instance no single user should draft, approve and effect payment with respect to a cheque. This section will address the issue of separation of duty in the context of our formulation. We demonstrate that using the O-O model principles of chapter 3 and the role organization structure of chapter 6, we can enforce separation of duty.

7.5.1 Separation of Duty & The O-O Paradigm

Separation of duty is a major requirement for commercial database integrity [CW87, Kar88, Lee88, San91, Tho91][2].

[2]Others are requirements such as all database items being constrained data items (CDIs), that only certified transformation procedures (TPs) manipulate the CDIs, that the TPs are certified to

Separation of duty is applied where several people (or processes acting on their behalf) are required to perform a given task. Such a task would be broken into subparts which are then assigned to different people. Every individual is then required to perform (say) only one of the subtasks with the restriction that none of the individuals can perform more than one subtask. From example 7.1, separation of duty will bar a single individual from updating both SIGN_1 and SIGN_2.

The main idea of separation of duty is to ensure that no individual can initiate an action, approve the same action and (possibly) benefit from the action. Separation of duty aims to spread the responsibility for various processing steps across different individuals (or their proxies) and achieve dispersion of of authority across individuals that access database information.

Among the enforcement requirements in the Clark & Wilson Model [CW87] is that SSOs *must* associate TPs with users and the associated CDIs to form relationships of the form: $(UID, (TP'CDI_1, CDI_2, \cdots)), \cdots)$. Here UID is the user identity, TP is the transformation procedure in question and CDI is the related constrained data item.

To demonstrate that our formulation meets this requirement observe that:

1. Since every role in the database is secure, i.e. it has an associated access control list of the form $\{\cdots, id_p, \cdots\}$, where the id_ps are the identifiers of the users authorized to the role.[3]

2. User authorizations to system information are of the form:

$$(rname, \{(o_i, m_j), \cdots\}, \{\cdots, id_p, \cdots\})$$

with $rname$ being the role name, o_i the object, m_j is a method and $\{\cdots, id_p, \cdots\}$ is the role's access control list (see page 173).

3. A user's privilege authorization scope (see definition 7.9) computes the privileges authorized to a user. These are all the privileges specified in the roles to which a given the user is authorized. In our O-O environment, these are object-method pairs, i.e. (o_i, m_j). Hence for a given user, an authorization scope is of the form $(id, \{\cdots, (o_i, m_j), \cdots\}$.

take CDIs from one correct state to another, that there are verification procedures (IVPs) that assure the integrity of code which manipulates the CDIs, etc. In general, all data transformations are required to be designed as well-formed transactions (WFTs) (see section 2.4).

[3]Note that neither a user responsible for the administration of role authorizations nor one that designs/implements the privileges associated with a role can be be authorized to the role.

In the O-O paradigm, objects with the same behaviour and structure are handled as a class. Since methods are defined in the class (see chapter 3), they will operate on all instances of the class. Hence $\forall (m_q, o_1), (m_q, o_2), \cdots$ in a user's authorization scope, if $o_1, o_2, \cdots$ belong to the same class, we can express the relationship as $(m_q, (o_1, o_2, \cdots))$. Consequently, a user authorization scope can be expressed as $(id, \{\cdots (m_i, (o_j, o_k, \cdots)), \cdots\})$. This relationship is clearly similar to the one suggested in the Clark & Wilson model and quoted above.

4. Thus given an authorization scheme for a given system and the system users, we can compute, for each user, the associated authorization scope. Further, given that no user, other than those authorized, can access system privileges, the user authorization scope for each valid user must be unique for a given authorization scheme. Indeed, given that the enforcement of security is mandatory, no unauthorized user can execute a privilege in the system.

7.5.2 Separation of Duty & Object Histories

Another key requirement of separation of duty is the use of audit information to ensure that before subjects are allowed execution, they have not participated in the processing before. However, as Karger [Kar88] observes, searching for such information in the audit record can be very costly; there is need to avail this information in a form and "location" that makes for its ready availability. In our case, we use object history (see chapter 3) for this purpose. In our O-O model, object history is part of object state and hence part of the object itself. This enables each object (at least the CDIs) in the database to keep track of its own audit information. We introduce a history attribute of the object to record audit information.

The class structure must be defined to reflect the desired object structure to ensure that objects (at least those that require separation of duty) keep track of their histories. In our case, one of the constraints of the CDI class and its subclasses is that objects must keep track of their history. The history and its form depends on the processing constraints required for the particular objects. It has a value which is the audit information required for its processing. In defining a class, then, we not only specify that there be a history attribute but also its nature, i.e. the domain of its value. In general, such a value will be a sequence of events in which the nature of the event is determined from processing requirements.

This history provides no more information than can be found in the audit trail;

nor does it preclude the storage of the same information in the system audit record. It merely avails the same information in a form that supports performance improvement. Karger [Kar88] makes a similar observation.

To enforce separation of duty requires non-participation in the current history which is necessary, but not sufficient, to guarantee access at any execution stage [Kar88]. The final decision on whether or not to allow access must depend on authorization and any constraints imposed on access that may take into account both the history and any other required information to make such a decision. Karger [Kar88] makes similar observations regarding token capabilities for control of object access.

Our refined access strategy retains definition 7.10 but imposes a separation of duty criterion.

With $id \in \mathcal{ID}, r \in \mathcal{R}, o \in \mathcal{O}$ and $m \in \mathcal{M}$ we have:

$$\Phi_2(id, r, o, m) = \begin{cases} \text{true} & \Longleftrightarrow \text{ id} \in \text{ r.racl} \\ & \wedge \forall e_i \in o.Hist, \text{ id } \neq e_i.uid \\ & \wedge \text{ (o, m) } \in \text{ r.pset} \\ \text{false} & \text{Otherwise} \end{cases}$$

The condition id $\neq e_i.uid, \forall e_i \in o.Hist$ ensures prior non-participation for the current user in any previous event on the object.

For this history to be useful, method executions must either update the history attribute or be part of some transaction whose execution updates the attribute. Processing constraints must ensure that each permitted (or attempted) execution on the object utilizes the history and updates it.

Example 7.4 **Consider the cheque object of examples 3.1, 7.1 and 7.2 which, as defined, does not keep track of execution history. We introduce another attribute (HIST), to record audit information associated with the object. We redefine the type structure of page 39. The redefined class structure of CHEQUE is:**

```
Name:      CHEQUE;                            /* Name of the class */
Structure: {                                  /* The Structure of its instances */
           PAYEE: String,
           PAYEE_ID: String,
           AMOUNT: Currency,
           SIGN_1: String,
           SIGN_2: String
           HIST: SequenceofEvents             /* Attribute for object history */
                                  };
Methods: { clerk,supervisor }                 /* Method List for the Class */
```

Further, method executions must be redesigned to update this attribute on attempted execution.

```
Begin {Some Transactional Execution}
        ...
        on invocation of method (m);      /* m could also be some TE */
        check:= $\Phi_2(id, r, o, m)$;
        if check then
                begin
                        execute method;
                        update(o.HIST)
                end
        else update(o.HIST)
End {Some Transactional Execution}
```

□

In example 7.4, method execution is part of a transactional process that reads history information, uses it along with authorization information and updates the history. It is important that such executions be transactional in nature: the whole process must be executed *atomically*, it must be *correct*, it must execute in *isolation* with respect to shared data and its effects must be *durable* beyond the execution. This illustration is similar to what Ravi Sandhu [San91] terms *transactional expressions*. We do not address the manner in which these executions are structured and processed. It suffices (for now) to say that it must be transactional in nature.

Notice also that our formulation realizes *dynamic* separation of duty [NP90] in that all we care about is that the current access attempt is authorized and that the said user's participation is not in the object history.

Each object as seen from the previous section, keeps track of its (access) history, the nature of which depends on the application and the nature of the audit information necessary for the system audit function as it relates to the object. Object history, as defined in chapter 3, is a sequence of events and collectively, object histories constitute system audit information. A system audit trail can be seen as a sequence of events ordered according to time. Theoretically, this can be derived by ordering events associated with objects in the system by the time at which they occurred.

However, this may turn out to be a very expensive exercise in that all object histories would have to be ordered. To reduce the task of generating system audit information, it is possible to have the system audit updated along with the update of object history. The choice of whether or not to do t ..s simultaneously would depend on the application and the cost of accessing and updating the audit trail.

The system audit trail, like a bank account, has unbounded history. The length of the audit trail "history" to be kept would also depend on how long certain information must kept and would be system-specific.

7.5.3 Separation of Duty & The Role Graph

In this section we discuss how the role graph of section 6.5.2 can be used to enforce the principle of separation of duty [CW87, San91]. Separation of duty requires that subtasks of a task be executed by different individuals. This way, barring collusion, separation of duty guards against fraud. Hence an individual who has executed some subtask of a given execution sequence will be barred from executing subsequent subtasks in the sequence. In our formulation, object history is useful in determining which individuals have participated in the execution sequence.

To enforce separation of duty using our role graph of chapter 6, we use the concept of paths (see definition 6.9) from the role graph model. For a given path, we associate some execution sequence such that subtasks are associated with the lowest ranking role at which they can be executed. For instance given a path A→B→C→D and some task T with execution sequence $t_1 \prec t_2 \prec t_3 \prec t_4$, we can map this sequence to the path. Hence A, B, C and D will be associated with subtasks t_1, t_2, t_3 and t_4, respectively. Accordingly, we shall directly assign privileges associated with t_1, t_2, t_3 and t_4 to A, B, C and D, with corresponding methods p, q, r, and s, respectively (see figure 7.2).

From the semantics of the → role relationship, it follows that A, B, C and D can execute subtasks $\{t_1\}, \{t_1, t_2\}, \{t_1, t_2, t_3\}$ and $\{t_1, t_2, t_3, t_4\}$, respectively.

To model the sequence of execution steps, we use the concept of authority paths and keep track of which users have executed in roles with privileges pertaining to a given set of subtasks. Using this history as well as the role privileges, we can make assertions to govern continued execution of the processing. We term separation of duty associated with some path of execution as *in-path* separation of duty. Formally:

Definition 7.11 *In-Path Separation of Duty: Separation of duty is the enforcement of separation of duty along a given processing path.* □

Where the order of execution is important, we must impose a condition that ensures adherence to such order. Suppose the sequence of tasks is represented $T =$

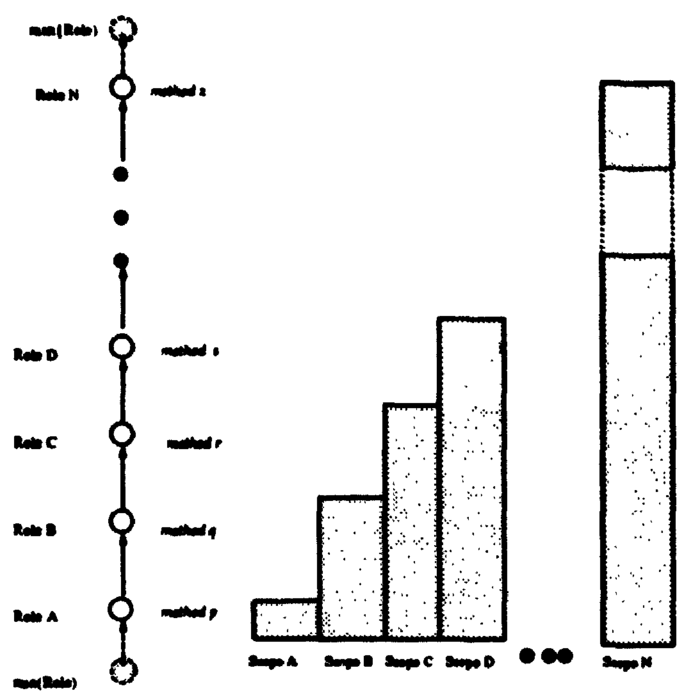

Figure 7.2: Distribution of Object Interface

$\langle t_1 \prec t_2 \prec \cdots t_n \rangle$. Let Φ_3 be such an access strategy (see definition 7.10) and let $id \in \mathcal{ID}, r \in \mathcal{R}, o \in \mathcal{O}, m \in \mathcal{M}$ and $t \in T$. Then:

$$\Phi_3(id, r, o, m, t) = \begin{cases} \text{true} & \Longleftrightarrow \text{id} \in \text{r.racl} \\ & \wedge \forall e_i \in o.Hist, \ \text{id} \ \neq e_i.uid \\ & \wedge \ (\text{o, m}) \ \in \text{r.pset} \\ & \wedge \forall (t_i \prec t) \in T, \ \text{Executed}(t_i) \\ \text{false} & \text{Otherwise} \end{cases}$$

Here t is the current task and t_i is some task preceding t in the sequence $\mathcal{T}$ and $Executed(t_i)$ is a Boolean that returns true when task t_i has been executed.

7.6 Conflict of Interest & The Role Graph

Conflict of interest can be seen as a form of separation of duty. However, while separation of duty usually refers to some given task, conflict of interest refers to separate tasks. This section formally presents the idea of conflict of interest and formulates a means of ensuring safeguards against it. We then proceed to demonstrate how we model it within our role organization model.

7.6.1 Roles & Conflict of Interest

An important requirement in commercial database integrity is managing conflict of interest. Conflict of interest is determined from an organization's conflict of interest policy which is part of an overall security policy. Conflict of interest exists between two roles if it is specified or implied so in the policy. For instance in the stock exchange there are regulations (with a force of law) that bar individuals with inside information regarding particular enterprises from trading in stocks of the same enterprises. Another example is our cheque and voucher tasks that would be in conflict should an individual responsible for approving a voucher also be the payee.

The principle of *conflict of interest* is applied where there is potential for conflict. Where it is employed, it is aimed at forestalling the potential negative impact of such a conflict of interest. To ensure there is no conflict, activities that conflict must be isolated and different individuals authorized to execute those operations which may conflict. In essence, this separates activities in conflict and ensures their separate processing. This differentiation of tasks and their separate processing has the result of reducing the potential for conflict.[4] Guarding against conflict of interest may be seen as a form of separation of duty. However, the use of the term applies to different activities whereas separation of duty pertains to sub-parts of one activity. For purposes of this work, we use the two terms in the contexts specified in this paragraph.

Two roles conflict if one user executing the two will violate the conflict of interest policy of a given security policy. Conflict thus is a matter of the system security policy. Such a policy would designate what activities conflict with which ones and ensure that assigning privileges to roles and defining role relationships do not cause conflict. Given a user u and roles r_i, r_j and a security policy $\mathcal{P}$, conflict can be defined as:

Definition 7.12 *Role Conflict: Role r_i conflicts with role r_j if a given user $(u \in r_i.racl) \wedge (r \in r_j.racl)$ while r_i and r_j have been declared to conflict by $\mathcal{P}$.* □

We term this the **conflict of interest constraint**: that no single user or group is authorized to two roles that conflict. Formally, we can define conflict

The effect of this constraint is to partition roles into what we refer to as *role conflicting groups*. A role conflicting group, CG, is a collection of roles without

[4]We say potential since, like security, conflict of interest is a human problem. Thus it may not be feasible to eliminate it totally.

conflict of interest among them. A user authorized to execute different roles in the group does not violate the conflict of interest constraint (see property 7.1). Formally, a conflicting group is defined as:

Definition 7.13 Role Conflicting Group, CG: *A role conflicting group, CG, is a collection of roles among which there is no conflict of interest, i.e. $\forall r_i, r_j \in CG_k$ there is no conflict while for $\forall r_i \in CG_k$ and $\forall r_j \in CG_l$ there is conflict since $CG_k \neq CG_l$.* □

In other words, given the universal set of roles $\mathcal{R}$, the conflict of interest relationship Υ partitions this universal set into non-conflicting sets. It is a mapping $\Upsilon : \mathcal{R} \mapsto \mathcal{CG}$ where $\mathcal{CG} = \{CG_1, \cdots, CG_n\}$ is a finite universal set resulting from the conflict relationship Υ. A given role in the system belongs to exactly one conflicting group:

Constraint 7.3 Role Conflict of Interest Constraint: *To ensure no conflict of interest, no role in a system can belong to more than one conflicting group.* □

It follows that for any two conflicting groups $CG_i, CG_k, CG_i \cap CG_k = \emptyset$. Consequently, a conflict of interest definition scheme is correct if and only if constraint 7.3 is observed.

A role authorization scheme in a system associates users with authorized roles. Without regard for conflict of interest such a scheme would be of the form $\Omega_1 : \mathcal{R} \times \mathcal{ID} \mapsto \mathcal{R} \times 2^{\mathcal{ID}}$ where $\mathcal{R}$ is the finite universal set of roles and $\mathcal{ID}$ the finite set of user identifiers. This scheme can be seen as generating an access control list for each role. An access control list is an element of the power set of the set $\mathcal{ID}$ (see definition 5.12).

Where a conflict of interest relationship Υ is defined, it must form part of the input for specifying access control to ensure that the authorization does not violate the *conflict of interest property* (see property 7.1 below). Hence such a scheme would be of the form: $\Omega : \mathcal{R} \times \Upsilon \times \mathcal{ID} \mapsto \mathcal{R} \times 2^{\mathcal{ID}}$ where $\mathcal{R}$ and $\mathcal{ID}$ remain as specified before and Υ is the conflict of interest relationship.

Property 7.1 Conflict of Interest Property: *Define $\Omega : \mathcal{R} \times \mathcal{CG} \times \mathcal{ID} \mapsto \mathcal{R} \times 2^{\mathcal{ID}}$ be some scheme that maps a given set of roles $\mathcal{R}$, conflicting groups $\mathcal{CG}$ and user identifiers $\mathcal{ID}$ to role-identier associations $\mathcal{R} \times 2^{\mathcal{ID}}$. Let r_i and r_p be some roles in conflicting groups CG_j and CG_q, respectively, and both to which some user with user with user identifier id_k is authorized where some conflicting relationship Υ holds. The mapping Ω is correct with respect to conflict of interest if and only if the two roles* *belong to the same conflicting group, i.e.* $CG_j = CG_q$. *We say the authorization scheme* Ω *has the conflict of interest property.* □

An authorization scheme $\Omega : \mathcal{R} \times \Upsilon \times \mathcal{ID} \mapsto \mathcal{R} \times 2^{\mathcal{ID}}$ is secure with respect to conflict of interest if and only if no single user or group is authorized to roles belonging to two or more conflicting groups.

Constraint 7.4 User Conflict of Interest Constraint: *No user can execute two roles from different conflicting groups.* □

Given a set of roles $\mathcal{R}$, a conflict relationship Υ, a set of user identifiers and an authorization scheme Ω, an important question to ask is whether Ω preserves the conflict of interest property. In other words, is there a user $id \in \mathcal{ID}$ with authorizations to two or more roles from different conflicting groups?

To solve this we use the concept of ***role authorization scope*** and the conflicting groups to ensure that no user is authorized to two or more roles in different conflicting groups. A user role authorization scope is the set of all roles the user is authorized to.

Definition 7.14 User Role Authorization Scope: *A user's role authorization scope is the set of all roles authorized to the user.* □

Algorithm 7.1 of figure 7.3 determines whether or not a given Authorization scheme has conflict of interest.

We determine the run time of the algorithm as follows

- Computation of CG takes O(m $\times$ n) with m roles and n conflicting groups.
- Computation of authorization scope AS takes O(m $\times$ p) for m roles and p users
- The rest of the algorithm takes O(m $\times$ p).

Hence we have overall performance of O(m $\times$ n) + O(m $\times$ p). In the general case the number of roles is much smaller than the number of users, i.e. $m < p$ and $n < m$. Therefore we O(m $\times$ n) + O(m $\times$ p) $\leq$ O(p $\times$ p) + O(p $\times$ p) $\leq$ O(p^2).

7.6.2 The Role Graph & Conflict of Interest

The basis of the role graph model is privilege sharing. It follows that roles with shared privileges have a likelihood of being authorized for the same user with the imposition of separation of duty. Hence such roles must be in the same conflict group. Hence

Algorithm 7.1 *Determination of Conflict of Interest:*

1. *Use the conflict relationship to generate the conflicting groups, i.e.* $\Upsilon : \mathcal{R} \mapsto \mathcal{CG}$,
2. *Identify authorized users. i.e. the set of users* $ID \subset \mathcal{ID}$ *and for all* $id \in ID$ *generate a role authorization scope. This results in pairs of the form* $(id, as) = (id, \{r_1, \cdots, r_n\})$ *where id is a user identifier,* $as \in AS$ *and* $as = \{r_1, \cdots, r_n\}$ *is its associated role authorization scope.* $AS = \bigcup as \subseteq 2^{\mathbf{R}}$, *on the other hand, is the set of all authorization scopes in the system.*
3. *For each authorization scope* $as = \{r_1, \cdots, r_n\}$ *and for each role* $r \in as$, *find the associated conflicting group.*
4. *There is a conflict of interest violation if and only if some authorization scope has roles belonging to more than one conflicting group. There is no conflict of interest if for each role authorization scope, every role belongs to the same conflict group. In other words,* $\forall CG_p, CG_q \in CG$ *and* $\forall as \in AS$ *if* $as \subseteq CG_p$ *then* $as_i \cap CG_q = \emptyset$, *for* $q \neq p$.

Figure 7.3: Conflict of Interest Algorithm

such roles with no conflict of interest can be seen as sharing privileges. The conflict of interest, on the other hand, precludes a user from executing two roles in different conflicting groups. Consequently, such roles, using our formulation <u>must not</u> share privileges. It follows that conflicting roles must be independent (see definition 6.13 on page 137).

Theorem 7.1 *Conflicting Roles & Role Independence: Conflicting roles must be independent.* □

Proof: From definition 6.13, two roles, r_i and r_j, are independent if and only if $r_i \odot r_j = \{\text{MinRole}\}$. We shall prove this by contradiction.

Assume that the two roles conflict but are not independent, i.e. $\exists \{r_k\} \neq \{MinRole\}$ such that $r_i \odot r_j = \{r_k\}$. Consequently, a user authorized to one of the roles can execute all the privileges of some $r \in \{r_k\}$. And since for such r, $\Psi(r) \subseteq \Psi(r_i)$ and $\Psi(r) \subseteq \Psi(r_j)$, it follows that the user will also be able to partially execute the other role, i.e. those privileges that are common to the two. However, this implies a violation of the conflict of interest principle. It then follows that for the two roles to be conflict-free, they must be independent. □

Theorem 7.1 holds as stated but not in the other direction. Hence independent roles may or may not conflict. In other words, even when the roles do not share privileges, they may or may not conflict. This is because the conflict relationship Γ is system defined and derives from the system's security policy. It thus requires that where roles conflict, then they must be independent. However, we can have independent roles that do not conflict where such conflict is not specified.

Therefore, from we have that conflicting roles must be independent and independent roles may or may not conflict.

Our task is to utilize the role graph model to manage conflict of interest and we do this by using the concept of role independence (see section 6.5.6). We aim at incorporating the conflict of interest constraints into the role graph model with theorem 7.1 being the guiding principle. This implies that each conflicting group can be associated with some independent partition of the role graph which has a direct relationship with MaxRole in the model. By ensuring that no user/group is authorized to roles in two or more independent partitions, we effectively preserve the conflict of interest property.

One can also use role independence to see whether there would be conflict of interest when given the role graph organization of roles. To do so, one must show that there exists one role in one conflicting group which is not independent of another role in another conflicting group. For n conflicting groups each with at most m roles we have a worst case performance of O(m $\times$ n) which is $O(n^2)$ in the worst case. This follows from the fact that if some role X is independent of another role Y, then Y is also independent of X.

7.7 Satisfying Commercial Security Requirements

The foregoing sections have outlined a formulation of integrity preservation with respect to the Clark and Wilson model [CW87]. While the intention has been to satisfy the requirements of the model, there has been no explicit demonstration how specific requirements of the model are met. In this section we address, in turn, each of the properties of the model and demonstrate how it is satisfied within our formulation.

7.7.1 Meeting Model Requirements

The model specifies two types of properties: certification and enforcement. We shall address these properties in the order in which they are presented in [CW87].

> *C1: All IVPs must ensure that all CDIs are in valid states at the time the IVP is run*

IVPs, essentially, act as audit programs to ensure the correctness and hence the integrity of CDIs. They serve to verify the effects of the TPs. For example in double-entry book keeping procedures an IVP would verify that for every credit, there is a corresponding debit and that any required balances are maintained. In our formulation, IVPs are designed to be transactional executions. They commit when correctness criteria are met for the associated CDIs. They abort whenever there is an anomaly and draw the attention of a user to that effect.

Like TPs, IVPs are designed with *pre* and *post* conditions which capture the integrity related conditions of the associated CDIs. In their execution, whenever IVPs return a negative results (i.e. an abort) they must return the nature of the error whenever the associated correctness requirements are not met.

> *C2: All TPs must be certified to be valid. That is, they must take a CDI from a valid state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a "relation" which defines that execution. A relation is thus of the form: $(TP_i, (CDI_a, CDI_b, \cdots))$, where the list of CDIs defines a particular set of arguments for which the TP had been certified.*

Operations, in our formulation, are defined using O-O principles. These, in turn, when invoked, operate on instances of the class in which they are defined. Let O be the set of instances of some class c with a set of methods M. It follows that $\forall m \in M$ and $\forall o \in O$, we have a relationship of the form (m, o) where method m operates on object o in some prescribed manner. Hence if we let $O = \{o_1, \cdots, o_n\}$ we have, for each method m, a relationship of the form $(m, \{o_1, \cdots, o_n\})$, i.e. method m operates only on the set of objects O and no other objects. This clearly defines a unique relationship between a method and instances of a class and is similar to the relationship specified in C2 above. The set of instances of the class specify a particular set of arguments that the method may manipulate. Moreover, the manner in which the method may manipulate the given set of instances is prescribed by the method definition.

Our formulation assures that each certified transformation procedure executed is guaranteed to obtain its input from named sources [GMP92] with such sources being the instances of the class in which the transformation is defined. The object undergoes a transformation as prescribed in the transformation definition within the class.

To ensure that each method leaves the objects it manipulates in a valid state, each method has associated pre and post conditions that must be satisfied before its effects are committed. These pre and post conditions are designed in such a manner which ensures the validity of the objects they manipulate. Consider a method that debits some account object which takes in the amount of debit, the current balance of the account and returns a new current balance. Assume that we must maintain a positive account balance at all the time of existence of an account object. This then defines the validity of an account. We can have pre-conditions that the (1) current balance is positive and (2) the current debit amount does not exceed the current balance in order to maintain a positive balance after update. The post conditions ensures that the committed action leaves the the associated object in a valid state. In our case, such a post condition would be that the balance must be positive. In a nutshell, each TP execution is subject to assertions regarding the condition of the CDI and the TP itself both before (pre-conditions) and after (post conditions) execution. Such assertions ensure that all CDI transformations are valid. Moreover, given that method executions are transactional, we are assured that all committed operations leave the objects in a valid state.

> *E1: The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulations of CDI is by a TP, where the TP is operating on the CDI as specified in some relation.*

Give our O-O approach each object is an instance of some class with specified methods for manipulation of instances of the class. Hence for some TP_i (a method in our case) we have a relation of the form $(TP_i(CDI_1, CDI_2, \cdots))$ where $CDI_1, CDI_2, \cdots$ are instances of the same class and TP_i is a method defined in the class. Given the manner of invocation of methods in the O-O paradigm, it is clear that no TP would manipulate a CDI other than those specified. Moreover, only prescribed TPs would manipulate any given CDI once invoked. Indeed, given that methods are invoked by message sending, in a purely O-O system, no other CDI manipulation is possible.

E2: The system must maintain the list of relations of the form:

$$(UID,(TP_i(CDI_1,CDI_2,\cdots)),\cdots)$$

which relates a user, a TP, and the data objects that TP may reference on behalf of the user. It must ensure that only executions described in one of the relations are performed.

In our formulation, each role is secure (see definition 5.13) such that:

$$\forall r \in \mathcal{R}, \exists \text{ r.racl}=\{\cdots,id_i,\cdots\}$$

A role authorization generates relations of the form:

$$(rname,\{(o_i,m_j),\cdots\},\{\cdots,id_p,\cdots\})$$

(see section 7.5.1). *rname* is the role name, o_i the object, m_j is a method and $\{\cdots,id_p,\cdots\}$ is the access control list.

Given an authorization scheme and some $id \in \mathcal{ID}$ we can generate a user's *role authorization scope* (the set of all roles to which the user is authorized) as: $(id,(rname_a,rname_b,\cdots))$. Substituting each role with its definition of the privilege list and rearranging the result yields a relation of the form $(id,(m_j,(o_1,o_2,\cdots)),\cdots)$. Since we regard methods as TPs and the objects o_is as CDIs, we have a similar relationship to that required by E2.

The list of relations in E2 must be certified to meet the the separation of duty requirement.

Our objects (at least those that require to maintain integrity) keep track of their histories (see section 7.5.2) in which an access strategy was defined. With $id \in \mathcal{ID}, r \in \mathcal{R}, o \in \mathcal{O}$ and $m \in \mathcal{M}$ we have:

$$\Phi_2(id,r,o,m) = \begin{cases} \text{true} & \iff \text{id} \in \text{ r.racl} \\ & \wedge \forall e_i \in o.Hist, \text{ id } \neq e_i.uid \\ & \wedge \text{ (o, m) } \in \text{ r.pset} \\ \text{false} & \text{Otherwise} \end{cases}$$

The condition id $\neq e_i.uid, \forall e_i \in o.Hist$ ensures prior non-participation for the current user in any previous event. This effectively imposes separation of duty on relations specified in E2.

E3: The system must authenticate the identity of each user attempting to execute a TP.

Each method is executed by users authorized to the associated role. Once it is ascertained that the user is authorized to the role, then such a user can execute any privilege in the role's privilege set. From the access strategy above, this authentication condition is satisfied by the condition id $\in$ r.racl.

C4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

Objects in our formulation keep track of their histories whose nature is determined by the application at issue. The nature of the history of an object would be application specific; it must be relevant information necessary to fulfill an audit function relating to the object. The object history is an append-only object that cannot be altered once created.

While it is possible to generate the audit trail by ordering the events in all object histories in a given system, this clearly would be too expensive to do. Consequently, it is necessary to update the system audit trail along with object history update (see section 7.5.2).

Consequently, the object history update operation in example 7.4 would be modified to appear as follows:

```
on invocation of method (m)
check:= $\Phi_2(id, r, o, m)$;
if check then
        begin
              execute method;
              update(o.HIST);
              update(system audit)
        end
else update(o.HIST)
```

Note that, like object history that is application dependent, system audit trail woul' also be system dependent.

C5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.

E4: Only the agent permitted to certify entities may change the list of such entities associated with the entities: specifically, those associated with a TP. An agent that can certify an entity may not have any execute rights to that entity.

Entity certification, like any activity, is associated with system privileges. These are defined in some role associated with certification. To execute a TP associated with certification, an agent must be authorized to the associated role and must satisfy the access strategy condition specified in section 7.5.1, both in terms of authorization and separation of duty.

The second condition of this enforcement property is met via the *conflict of interest* conditions. Entity certification and execution are clearly conflicting operations and would belong to different conflicting groups. Since no user can be assigned roles from different conflicting groups, it follows that no user would be authorized to roles pertaining to both execute and certification of a given TP.

7.7.2 Security Information Management

Management of security information is as important as ensuring proper authorization. Security information management entails the specification and maintenance of the information pertaining to authorization information and enforcement. Hence role definition, modification, authorization and all information pertaining to information access falls under the management of security information. In general, the security management function can be seen as pertaining to facilitating authorization, authorization information and its administration which facilitates ensuring system security, authentication and its accuracy. It involves both the management of associated pieces of information as well as the correctness of their administration.

In our system, the security information is distributed across different roles. The authorization and execution of a role pertaining to security information is not any different from that of any role. However, correct emphasis must be placed on the possibility for conflict. For instance, two roles pertaining to the execution and administration of some piece of security information must be treated as conflicting.

Hence no one user can be authorized for both the administration and execution of a given role. Moreover, no user is authorized to specify authorization information pertaining to the user's authorization. This strategy then allows us to "watch the watchers".

In general, such requirements on the administration of security information is part of system security policy which is of the form:

1. Every execution must be due to invocation by a duly authenticated and authorized user. Hence every execution must be authorized.
2. Every authorization must be via *user-role* authorization.
3. Every role must be duly defined in terms of specified privileges.
4. Every privilege in the system must be a *valid* access modality of the associated object.
5. Each object must satisfy C-O requirements including being an instance of a class, accessible via methods of the associated class, must have a unique identifier, must have *history* as a component of its object state, etc. Other requirements include the specification of methods as TPs, mandatory object history update whenever an object is accessed, etc.
6. All specified security constraints must hold at all times in the life of a system.
7. Object access must be subject to separation of duty.
8. The system must enforce a policy that forestalls conflict of interest.
9. The system must meet the requirements of the Clark and Wilson model for commercial security.

7.8 Summary & Conclusions

This chapter has demonstrated how our formulation meets commercial security requirements outlined in the Clark and Wilson model for integrity. Using an object model, a transaction, a formal role definition and role organization, we were able to formulate a scheme that assures commercial integrity.

We used O-O concepts to define roles. Methods associated with a given object are assigned to different roles and thus offer differentiated access to object information.

This strategy which is similar to that of [TDH92], enables us to utilize a given object interface to manage access control to the object information. Imposing a transaction framework on the object interface enables us to exploit the transactional properties of the model of chapter 4 which further allows us to model properties of well-formed transactions as stipulated in the Clark and Wilson model.

Separation of duty is another major requirement of commercial integrity. In our formulation, we introduced object histories as one way of aiding in enforcing separation of duty. Object history is important for determining which user has participated in any task that is under process. Further, our formulation demonstrates that we realize the kind of TP-CDI relationships specified in [CW87]. These requirements are met via the class-instance and class method relationships in the O-O paradigm. Further, with respect to the user-TP-CDI relationships, we showed that we can generate equivalent relationships from user-role authorizations and associations between the roles and the methods in question.

Using the role graph of chapter 6 we demonstrated how paths in the graph can be associated with processing sequences such that a task in the sequence is associated with some role in a given path. Related to this is what we termed *in-path* separation of duty.

Yet another important concept discussed in this chapter is that of conflict of interest. We formally defined the concept and demonstrated how to guard against it. We showed how our role graph can be used to determine what role assignments preserve the conflict of interest property. This property expresses the fact that no role can belong to two conflicting groups and that no user must be authorized to roles from different conflicting groups. For instance TP certification and TP execution clearly must belong to different conflicting groups. Hence no user would be authorized to roles that involve both certification and execution. It has to be one or the other or none.

With respect to our role organization model, the conflict of interest requirement results in a partition of roles into independent groups whose least upper bound and greatest lower bound are MaxRole and MinRole, respectively. In other words, the conflicting groups of roles must have no coupling.

Among the key contributions of this chapter include a demonstration of the realization of principles of the Clark and Wilson model within our formulation. Among these are the separation of duty using object histories, the generation of TP-CDI relationships from O-O principles and user-TP-CDI associations via user-role authorizations, class-method and class-object associations also from the O-O paradigm. Object histories are defined as part of object structure in our model while class-method and class-object relationships are directly defined from class definitions and method invocation semantics in the O-O paradigm.

Modeling conflict of interest using role independence in our role organization model is another major contribution of this chapter. We formally defined and adapted the conflict of interest property to our role organization structure.

CHAPTER 8

Summary, Contributions & Future Directions

8.1 Introduction

This thesis focused on commercial security and its realization using O-O principles. Commercial security emphasizes the integrity of data and procedures that manipulate the data objects whereas traditional security (as in government and military systems) emphasize both secrecy and integrity. In commercial security, the accuracy with which a bank account is maintained is more important than the secrecy concerning the value of the account. The basis of the Clark and Wilson model [CW87] which lays out the properties of commercial integrity, is the operation-data relationship. Data objects have defined valid states and operations which manipulate the objects are specified in such a manner that assures valid output state for the objects they manipulate. The choice of O-O modeling arises from the observation that there are unique operation-data relationships defined via the class concept. Hence to ensure the correct manipulation of data objects we define the methods in such a manner that they have properties which assure that the objects they manipulate will be guaranteed to be in valid state.

A substantial portion of this work was devoted to the role concept, its definition and application in the management of access rights. The role concept is implied in the Clark and Wilson model, hence we endeavoured to formalize it, demonstrate its use access rights management as well as the administration of roles. Other interesting investigations included the demonstration of the strength of role-based protection.

The current chapter will summarize the work covered in this thesis, discuss what we believe to be the key contributions of this work to research in security and indicate possible directions for further research. In the next section we summarize the contents of this thesis. Section 8.3 discusses the major contributions of this work while section 8.4 addresses outstanding issues for possible investigation in the future.

8.2 Summary

Chapter 1 defined the problem addressed in this work, set out the goals of this thesis and the manner of tackling the problem to achieve the set goals. The **key goal** was the formulation of an O-O based realization of commercial security as specified in the Clark and Wilson Model [CW87]. Consequently, it was necessary to address the issues associated with the Clark and Wilson model, albeit as a summary. This formed the subject of chapter 2.

O-O principles offer a powerful means of modeling of complex objects and their behaviour and hence their suitability in our current application. Hence given that we had intentions of using O-O principles in our formulation of a model for commercial integrity, we developed an object model for the purpose. We found this task necessary given the lack of a standard O-O model. Indeed, without a model and a clear definition of the O-O terms used, there would be risk of multiple interpretations. The specification and formulation of an object model was the subject of chapter 3. Of particular importance to our problem is the object interfaces and the definition and manipulation of the database objects. We developed a method execution scheme which can be represented by a method execution tree. A formal definition of an object interface was necessary since it is via the interface that objects are manipulated; it is via this interface that object integrity can be assured. Yet another important tenet of our object model is the concept of object history that is necessary for assuring separation of duty in commercial security. It is important that such history be part of an object.

Given the flexibility of defining and composing of O-O objects, we found it appropriate to incorporate such history within the object structure. As such, every object (at least all those that must maintain their integrity) must keep track of their histories. Such history is necessary for enforcing separation of duty; making it part of the object structure makes it readily available. The aim is to save valuable processing resources that would be used in extracting the same pieces of information from the system audit trail.[1] With the O-O concept of extensibility, it was possible to incorporate a new type in the type hierarchy with the desirable properties such as object histories and transactional interfaces.

[1]Karger [Kar88] achieves the same via use of token capabilities.

From the requirements of the Clark and Wilson model [CW87], one requirement is the transactional execution of transformation procedures. Hence it was necessary to formulate a transactional model within which all executions would take place. This was necessary given that there is no standard nested transaction model, just as there is no standard O-O model. Our transaction model of chapter 4 extended the O-O method execution of chapter 3. This involved the incorporation of transactional properties in the method executions specified along with the object model. The effect of this was to make object interfaces transactional. This has the advantage that operations on objects would have *all* or *none* of their effects causing change on the objects they act on. This is important in commercial integrity for instance in the case of double entry procedures. If only one such entry were effected, clearly, the integrity of the data would be questionable. Nesting arose as a natural consequence of the method execution tree of chapter 3. The formulation of a transaction model was the subject of chapter 4.

The role concept is an integral part of commercial security and was the subject of chapter 5. From the related work, there is no formal notion of the role concept (except perhaps Baldwin's [Bal90] named protection domains). Consequently, we found it necessary to define it formally. Roles are defined as collections of privileges. Hence a formal privilege definition was necessary. Other issues discussed with respect to role formalization, were the advantages of using role-based protection which are mainly associated with flexibility in access rights administration. The shortcoming of the approach lies in the complexity associated with the analysis of the implications of privilege assignments. A formal role organization scheme simplifies this complexity a great deal.

In studying roles and their application to role-based protection, we went beyond mere role definition in order to understand its strengths. As such we formulated an information flow scheme based on roles. In this scheme we proposed a methodology that would ensure that roles defined in a system would not violate system security as specified in the security policy. The approach ensures that any flows resulting from role definition do not contravene the information flow scheme derived from the security policy. The solution chosen uses graph theory and in particular the subgraph isomorphism strategy for determining role definition consistency with a given security policy. This yields a complexity of $O(n^2)$ where n is the number of roles in the system.

Yet another important issue addressed with respect to roles and traditional protection schemes is the application of roles to realize traditional mandatory access control. Using information flow analysis strategies, we demonstrated that role-based protection can emulate mandatory access control. By considering role update and read scopes, we formulated rules that will ensure that mandatory access control-like security can be achieved. In doing so we demonstrated that role-based protection approaches to system protection can be comparable to traditional schemes (in this case mandatory access control).

As hinted earlier a formal role organization scheme can ease the complexity of access rights administration. Role organization was the subject of chapter 6. The basis of the role organization model is role relationships and by examining role definition and examining these basic role relationships, we were able to extract properties that must be ensured in a role organization model. Among these are the acyclic nature of the graph model that represents role organization and the privilege monotonicity property for roles in any path in such a graph. Along with the role organization we proposed algorithms for role administration in role-based schemes. These algorithms could be used to realize a role management tool which preserves the properties of the role organization model.

Chapter 7 demonstrated the realization of commercial integrity within our formulation. Using the object model of chapter 3, the transaction model of chapter 4, role definition and role organization model of chapter 5, we were able to offer a formulation which realizes commercial security as per [CW87].

Chapter 7 also addressed user-role authorizations, role-privileges authorizations and their means realizing access control. We demonstrated the relationships between constrained data items, transformation procedures and system users. We showed how relationships specified for commercial integrity can be computed from those specified via user-role and role-privilege authorizations. Another important aspect addressed was the use of object histories in enforcing separation of duty.

Conflict of interest was also discussed and a distinction was made from separation of duty. Using the role organization model, we demonstrated a means of guarding against conflict. This led to the concept of role groups and whether or not they conflict. Conflicting roles must be associated with different groups and must be independent of each other in the role organization model. Indeed, conflicting groups can be seen as partitions of the role organization model into independent groups of roles.

8.3 Key Contributions

The original goals included formulating commercial integrity using O-O concepts. However, as we have shown in the summary, this work also encompassed other associated concepts such as roles definition and organization, information flow analysis and roles as well as emulation of mandatory access control in role-based protection systems. This section will summarize what, in our view, are the major contributions of this research. We shall enumerate them in no specific order of merit.

1. **Formalized the Role Concept:**

   The literature has no formal role definition. The term role has been used widely but its context varies as widely as its usage (see for example [RWBK91, San91, Tho91, DM89, Bal90]) leading Baldwin [Bal90] to choose the *named protection domain* in order to avoid potential confusion. Therefore, we believe that a formalization of the role concept will go a long way to clear up this apparent confusion. This work presented a formal definition of what we mean by role. In doing so, we used another important concept whose formalization we realized as well. Accordingly, we defined the term privilege.

   A privilege was defined as a pair: an object and an access mode. It specified what can be done by a user authorized to the use of the privilege. Our privilege definition is general enough to be subjected to any application specific security policies. For instance a capability is a special case of a privilege where a user can copy and distribute the capability according to need, albeit with no possibility of altering such a capability. The flexible nature of the privilege allows us to define it in any suitable form based on application needs. As an example, the definition of privilege on the basis of O-O object interface is a direct consequence of its rather loose definition. In a word, the stringency of conditions that determine the nature of a privilege can be varied to suit application requirements.

   Having defined a privilege, we defined a role as a collection of privileges. The flexible nature of privileges means that we can define roles with similar flexibility; the nature of role will depend on the nature of the associated privileges in its privilege collection. It is this flexibility of role definition that allows us to use O-O principles to define roles. By specifying privileges as partitions of an O-O object interface, we are able to formulate a role definition based on O-O interfaces. What is more is that such interface can be made transactional, a further testimony to the flexibility arising from privilege definition.

Our formulation was not confined just to role definition and formalization. We went further and studied role-based security and its associated benefits in system protection. Consequently, we studied the advantages and shortcomings of role-based protection schemes. Among the advantages include its in system access rights administration. The flexibility arises from the apparent two-stage authorization process used in role-based protection: user-role authorization and user-privilege authorization. Flexibility is gained via the ability to assign and revoke access rights associated with a user. This can be achieved via privilege assignment to a role or via revocation of user authorization to a role. This flexibility is important considering that security constraints tend to make system access rights administration very inflexible. A down side of this protection approach is that there is apparent complexity when it comes to the analysis of the assignment and implications of assignments of system access rights. It is inherently more difficult to analyse access rights assignments in such a scheme compared with tradition schemes such as mandatory access control.

2. **Roles, Information Flow & MAC:**

   In studying role-based protection further, we explored information flow analysis in such environments. In discussing information flow in role-based security we developed a means of ensuring that role definition schemes preserve system information flow policy. To enable us answer whether a given role definition scheme preserves the associated system security policy, we proposed a graph algorithm that compared flows resulting from role definitions and those emanating from the security policy to determine whether the two are consistent. A system is secure if the information flow scheme arising from role definition is consistent with the information flow policy.

   Yet another important issue studied following the formalization of the role concept, is the demonstration of the ability to realize mandatory access control using role-based protection. This aspect of role-based security is important as it establishes the strength of this mode of protection.

3. **Proposed a Role Organization Model:**

   As hinted in the previous item of contributions, one drawback with role-based security is the complexity associated with access rights administration in such systems. Thus it is important to organize roles in such a manner that this complexity is minimized. Therefore, it was important for us to extract properties of roles that would facilitate the formulation of an organization model. The basic focus of such search of properties is in basic role relationships. We found that roles can be related via partial, common and augmented privilege sharing. While these may appear distinct, they are all related via the partial role ordering relationship. For purposes of modelling we introduced the concepts of maximum and minimum privileges in a system. Combining all these relationships resulted in a ro'e organization model which is basically a directed acyclic graph.

Having proposed this model, we demonstrated that it had similar expressive power as other role organizing structures such as hierarchies , named protection domains and Ntrees. Essentially, our model has similar properties (acylicity and privilege monotonicity in a path) as the other structures.

For purposes of role management, we proposed a set of algorithms that would facilitate role management. These algorithms pertain to role deletion, addition and partition (both vertical and horizontal).

4. **Formalized Commercial Security Using O-O Concepts:**

   The choice of O-O concepts for formulating the Clark & Wilson model, arises out of the observation that the model is operation based, i.e. the model . ~quires that there b~ unique relationships between data objects and operations on the data. The O-O paradigm provides such a relationship via the class concept in which methods which operate on instances of the class are defined. Moreover, given that methods can be tailored to realize the effects of desirable operations that we need, O-O approaches offer great advantage. Further, we know that we can also specify the methods in such a manner that they meet the required conditions of the TPs in the formulation. Hence the choice of O-O modeling.

   Using the role concept and O-O approaches we realized the principles of the Clark and Wilson model for commercial security. In particular we introduced the concept of interface partition and distribution among roles. Object histories (see next item) were proposed for the enforcement of separation of duty. By defining the interfaces as WFTs and the objects as the CDIs, we guarantee the realization of principles of the Clark and Wilson model.

5. **Object Histories:**

   Object histories were introduced as a means of keeping track of audit information pertaining to objects. Like Karger's [Kar88] capabilities, our object histories ar~ intended to simplify the task of extracting such information from (say) a system audit trail. We defined history in terms of events where an event represents some action pertaining to object access. The event was defined in a general form and can be specialized to suit a given application depending on the nature of audit information required. Given the extensible nature of O-O modeling, it was possible to incorporate object history as part of the object information. Whether defined as an object attribute (and hence as part of object state) or as a separate component of object definition, is a matter of an application. Our demonstration used history attributes for this purpose.

   With respect for commercial integrity, object histories are useful for keeping track of audit information and for the enforcement of sep[illegible]on of duty.

Finally, we point out that much of our research has focused on and utilized a number of ideas already pursued in computer security, object-orientation and database research. Part of our achievement in this work has been to bring together known concepts such as roles, transactions, mandatory access control, commercial integrity, information flow, object-orientation, etc. into one formulation. Thus we examined the role concept and gave it a formal definition; we used information flow techniques to propose a means of ensuring conformance of a given design to a given security policy; we used information flow techniques to demonstrate the soundness of role-based protection and how it can realize a level of protection equivalent to that of mandatory access control; we used object-orientation to capture object histories and applied transactional properties to executions. Once combined in the manner demonstrated in this work, we realized the concepts of commercial integrity.

8.4 Future Directions

In the foregoing section we have outlined some achievements of our research. The value of research, however, is not just the results it demonstrates but also the "doors" it opens for further investigations. In this section we outline some of these directions which this line of study could take.

1. **Implementation & Experimentation**

   There is a need to implement a role-based prototype that would facilitate further study of the results arrived at in this work. Such a tool would be useful both for the investigatigation of the applicability of theory arrived at in this thesis. Some rudimentary Pascal-based implementation of the role graph algorithms was done. It did offer some insight on some practical issues of such an implementation. However, a more robust system needs to be implemented and used in experimentation for access rights administration. This will give a clearer picture on how complex the administration becomes in real life. Further, insights may be gained into what other requirements are essential for a successful role-based application. Indeed, the need for persistence of role-definitions and the dynamic nature of addition and revocation of access rights suggests database storage schemes with fast access.

2. **Role-Based Protection & Constraints**

   There is a need for a study of users, roles and resources along with the effects on security due to imposing constraints on their relationships. For example suppose that, like role relationships, we had user and resource relationships? Or suppose that we had some constraints (say) that impose restrictions on concurrent access of user/user groups to resources/resource groups? Such a study would delve into the nature of the constraints and how to express them. No attempt was made in this direction including the manner of expressing and managing such constraints. A probable future study is to formulate a language of formally expressing the said constraints and a means of constraint checking to assure their consistency.

   It may be worthwhile to explore the suitability of expressing such constraints using constrained logic and applying constraint consistency checking.

3. **Roles & the Relational Model**

   One reason that the O-O model was suitable for our formulation is that the complex operations/computations can be captured via the class concept where all operations are defined which are applicable to instances of the same class. Defining roles within the relational database paradigm presents a totally different approach. For one, we must talk about relations in first normal form in which table entries are simple attributes. Moreover, talking about operations, we must talk about simple reads and writes. How then do we configure roles within this framework?

These are but a few of the directions future work in this area could take. However, there are possibly more directions that we have not determined currently but which we or some other interested researchers could undertake following the results presented here.

APPENDIX A

Related Work

A.1 Introduction

In this appendix we summarize work related to our own. The basis of our work is the Clark and Wilson model for integrity [CW87] that was proposed for application in commercial environments. As emphasized in the body of the thesis, the main focus of this work is the integrity of information. Several approaches have been proposed for realizing the properties of this model. Among these approaches are the role-based [San91], secure capabilities formulation [Kar88], and the mandatory integrity approach [Lee88].

Roles, on the other hand, given their advantages, have been used in various environments such as in an object-oriented (O-O) software design environment [TDH92], to realize commercial security [San91, Tho91] and in a medical delivery system [Tho91]. We shall summarize discussion on roles and commercial database security in section A.2. Specifically, we discuss roles and commercial security in section A.2.1, roles in an O-O design environment in section A.2.2 and roles as used in a medical delivery system in section A.2.3.

Secure capabilities and mandatory integrity formulations have also been used to realize the Clark and Wilson model. These are the subjects of sections A.3 and A.4, respectively.

It is important to note that the Clark and Wilson model implies role-based protection, though it does not say so explicitly. Hence even when we talk about its enforcement using mandatory integrity or secure capabilities, roles still lurk in the background.

Ting et al. use roles for specifying and managing access rights in an O-O design environment. They propose a hierarchical structure to capture role organization.

Section A.2.2 summarizes this application.

Domain definition tables, DDTs, have also been used for role definition [Tho91]. We present the summary in section A.2.3.

Roles have also been referred to as named protection domains [Bal90] and organized into privilege graphs (see figure 6.2). This is the subject of the discussion in section A.2.4.

A.2 Roles & Role Organization

A.2.1 Roles & Commercial Database Integrity

The Clark and Wilson model [CW87] is one application where the need for high level integrity is desirable. In this model, users, programs (well formed transactions) and system objects (constrained data items) are associated to enable the specification and enforcement of integrity [CW87, Tho91]. Well formed transactions (WFTs), which act on system objects (constrained data items, CDIs) are associated with roles. Users are, in turn, authorized for the defined roles. Such authorization facilitates access to objects associated with the WFTs in that role.

Roles themselves can be organized and administered using any of the structures suggested in section 6.2 while constraints can be specified in the execution of the WFTs. For instance to enforce separation of duty [CW87] it may be specified that continued execution, of any execution order, be dependent on previous history. Since separation of duty requires that no individual can perform more than one function in the history of some processing, such execution would keep track of the identities of users that have participated in the execution so far. The principle of separation of duty can be enforced in role-based applications using different strategies. Sandhu [San91] suggests use of *transactional expressions* to achieve this and extends the original Clark and Wilson model by accommodating dynamic *substitution of attribution.*

A.2.2 Roles & O-O Design Environment

T. C. Ting et al. [TDH92] propose a user role-based protection approach to an O-O design data model for software engineering environments. Their approach, like that of Rabitti et al. [RWBK91, RWK88] in the design of a an O-O protection model, is to exploit the O-O paradigm to organize user roles, define their capabilities and offer a framework for analysing and understanding the implications of using user role-based protection. Users are organized into user groups and the groups have their interrelationships defined by some specified "ordering". To determine what information needs to be available to different groups, different methods are assigned to different user groups. Effectively, method assignment determines what portion of an object's (or class') public interface is made visible to different groups. This way, the object oriented principle of information hiding is extended and exploited. Extended in that no single user, unless authorized, need execute all methods of a given type. Exploited in that any given object of some type has its interface effectively "windowed" to provide different access to different groups.

The approach advocated by Ting et al. offers a framework for organizing users (see user role definition hierarchy, URDH, below) in a system, determining their needs which, in turn, determine the capability of the users through the roles authorized to such users. Capability assignment is realized through method assignment to the individual roles. Since methods are the only means of access to objects in an object oriented environment, methods not assigned to a user's role effectively hide the information they access from such a user.

Suggested is a framework for defining and analysing the implications of authorizations in a system. The task of defining user capabilities is a two-stage process. First, a URDH used to define possible user roles in a particular environment is instantiated. The URDH not only defines what roles are necessary but also captures the relationships between them. Second, capabilities are assigned to individual roles, a task realized via method assignment. Using relationships between the defined roles, an appropriate method assignment scheme is achieved that makes the least number of role-method assignments to achieve desired functionality. Such analyses can detect capability over provision (an inconsistency) and under provision (a form of incompleteness).

Ting et al.'s work also discusses techniques for analysing role capabilities and the objects they pertain to in a given URDH. These techniques provide a means of assessing and/or identifying any conflicts that may result from an initial URDH method and object type instantiation.

A.2.3 Roles in a Medical Delivery System

Another example is a medical delivery system suggested by Thomsen [Tho91] in which DDTs are used to specify user privileges. Given the diverse nature of medical delivery services and the varied needs of users, roles offer a handy means of system capability management.

The DDT approach is based on the principle of *type enforcement* [Tho91] in which every system object is assigned a *type* while each subject is assigned a *domain*. A DDT is defined to show which type of access subjects of a given domain can have on objects of a given type. These entries, from a type enforcement point of view, may be seen as classes of subjects (domains) and objects (types) with the specified access type associating the two [Tho91]. Associating a user to a domain determines the user's authorization. Such a user has the type of access specified for the related domain to the objects of the associated type.

To ensure a mandatory policy enforcement, type enforcement is made a system function in which neither subjects, objects nor processes acting on their behalf can alter their authorization attributes (domain and type memberships). To achieve role-based protection, each domain can be viewed as a role. The access rights (of some user from some domain) to a given object are determined by the type of access the associated domain has to the type of the object.

As indicated elsewhere, the major distinction between DDTs and the other structures (lattices, Ntrees, hierarchies) is that DDTs do not capture the the structural composition of roles [DM89]. Hence role inter-relationships must be specified outside the DDT itself. As well, unlike any of the other structures, DDTs also cannot specify authorization information within themselves. While this has not been stated explicitly, the role functional and structural specification can be specified to capture the access control information when it comes to hierarchies and lattices. In a nutshell DDT do not prescribe the flow of authority in the system they are modelling and this is their major shortcoming.

A.2.4 Roles as Named Protection Domains

Baldwin [Bal90] terms roles **Named Protection Domains** (NPDs).

Baldwin [Bal90] organizes roles, which he terms *named protection domains*, into privilege graphs (see figure 6.2). The privilege graph has three types of nodes: user, named protection domain (or role), and a functionality. Each role has an associated functionality. A user authorized to a given role has an arc connecting the user node to the associated role. A role's relationship with a given functionality is captured by an arc from the role to the associated functionality. An arc from role X to role Y implies that the privileges and functionality of role Y can be executed by an authorized user to role X. Consequently, a user is authorized a given functionality if there is a path from the user node associated with the user to the functionality in question. The cumulative privileges available to the user, which we shall define later as the *user's privilege scope*, is the set of all privileges in the path from the user's node to all the functionalities. One can infer that the user's authorization scope in the graph is a tree.

A.3 Capabilities & Commercial Integrity

Karger [Kar88] discusses the realization of the Clark and Wilson Model [CW87] using *secure capabilities*. A capability as used here facilitates access to database objects. Possession of a capability in the formulation is necessary, though not sufficient, to guarantee access to a database object. In formulating a realization of the CWM model, Karger notes that the problem of verification would be a daunting and suggests that attempts must be made to ensure commercial security without requiring verification for every piece of code.

The concept of separation of duty [CW87, San91, Tho91] suggests *history dependent* access control. Such history can be gleaned from audit trails. However, the task of searching the trail for the necessary information c : be a daunting and time consuming task. Audit trails are large and complex making it prohibitive to search. Hence the author's solution is the use of what he terms *token capabilities*. Token capabilities keep track of audit information of the associated object. Such information would include the identity of the subjects that have accessed the object to date and any other information required for realization of separation of duty. Token capabilities thus avail audit information much more easily and its ready availability (as opposed to audit trail search) facilitates performance improvement. Access to an object is then dependent upon access information (say) from some access control list, audit information captured in the token capability and the rules governing access based on these pieces of information.

Token capabilities can also be used to handle groups of users, as opposed to a single user by use of operating system (OS) access control principles. For instance by grouping users into the *owner*, *group* and *other*, one can specify access rights of users based on their group memberships.

This work also discusses the integration of audit, security and recovery strategies (e.g. two-phase commit and time stamping) to ensure commercial integrity.

A.4 Mandatory Integrity & Commercial Security

Lee's [Lee88] approach is to enforce commercial database security as specified in by Clark and Wilson [CW87] using mandatory integrity, specifically, the use of integrity categories. Lee contends that traditional models such as Bell and LaPadula's [BL75] have natural extensions that can realize the principles of the Clark and Wilson model [CW87] and goes on to enumerate the rules necessary for the formulation.

Two main ideas are employed: *mandatory integrity categories* and *partially trusted subjects*. The former is used to control unauthorized modification (the key concern of commercial security) while the latter, based on a variation of the Bell and LaPadula model, is used to represent and control authorized transactions.

A trusted subject is assigned to integrity labels: *view-min* and *alter-max*. The former refers to the minimum category that the subject can view while the latter refers to the maximum category of information that the subject can write/alter. An untrusted subject has *view-min=alter-max*. Two rules govern access to objects based on the integrity label:

1. **Simple Integrity Condition:** a subject can write an object o if label(o) is a subset of alter-max.

2. **Integrity *-property:** A subject can read an object o if view-min is a subset of label(o) or alter-max is a subset of label(o).

An *integrity category* is a name of a particular type of protected information/data where a given user (or program acting on the user's behalf) cannot *create* or *modify* any given type of information/data without *explicit* authorization for that type. A set of categories, as applied to *untrusted subjects* (users, programs, etc.) indicates that the subject is authorized to create/modify information/data of any type in that set and not others. Of course a subject can have authorization to more than one category! A *partially trusted subject*, on the other hand, is allowed to transform a limited set of input types to a limited set of output types such that the inputs are at least view-min while the outputs must be at at most alter-max. Used this way, integrity categories facilitate the limiting of a subject's ability to modify information.

Lee observes that the Clark and Wilson model essentially reduces to identifying the required roles in a system, determining conflicting roles and ensuring that no user acts in two or more conflicting roles, and letting each specific kind of application data/information be modified by specific approved transactions acting on behalf of some user executing in an authorized role.

To realize the Clark and Wilson model, Lee then proposes eight rules that must be adhered to. These include establishing the roles to be supported, identifying all different kinds of CDIs [CW87], specify administration roles that must be distinct from those specified in the earlier steps, distinguish conflicting roles and ensure that no user is authorized to two or more conflicting ones, install certified transactions and ensure that no unauthorized users can change them, establish transactions as partially trusted subjects and specify (for each) view-min and alter-max, establish security authorizing transactions, and ensure that consistency checking works correctly and that it is protected accordingly. Further, there is specification on *who watches the watchers* where it is specified that one (any user, program acting on user's behalf, etc.) cannot change one's security authorization, one cannot install nor modify transactions that they execute, one cannot install transactions outside one's jurisdiction , and that no one can delete nor alter audit records.

Lee goes to show that this formulation meets exactly the requirements of the certification and enforcement rules of the Clark and Wilson model.

REFERENCES

[AA92] D. Agrawal and A. El Abdi. Transaction Management in Database Systems. In Ahmed K. Elmargarmid, editor, *Database Transaction Models for Advanced Applications*, pages 1–31. Morgan Kaufmann, 1992.

[AAL+93] M. Abrams, E. Amoroso, L. J. LaPadula, T. Lunt, and J. G. Williams. Report of an Integrity Research Group. *Computers & Security*, 12(7):679–689, Nov 1993.

[ABD+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The Object Oriented Manifesto. In *ACM SIGMOD '90 Proceedings*, May 1990.

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal of Computing*, 1(2):131–137, June 1972.

[AH90] T. Andrews and C. Harris. Combining Language and Database Advances in an Object Oriented Database Environment. In S. B. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*. Morgan Kaufmann, 1990.

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, second edition, 1988.

[Bal90] R. W. Baldwin. Naming & Grouping Privileges to Simplify Security Management in Large Databases. In *Proc. 1990 IEEE Symposium on Research in Security & Privacy*, pages 116–132. IEEE Computer Society Press, May 1990.

[BB88] J. Biskup and H. H. Brüggermann. The Personal Model of Data: Towards a Privacy-Oriented Information System. *Computers & Security*, 7(6):575–592, Dec 1988.

[BCG+87] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, H. J. Kim, F. Manola, and U. Dayal. Data Model Issues for Object Oriented Applications. *ACM Trans. Office Information Systems*, 5(1):3–26, Jan. 1987.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object Oriented Database System: The Story of O_2*. Morgan Kaufman, 1992.

[BKK88] J. Banerjee, W. Kim, and K. C. Kim. Queries in Object Oriented Database Systems. In *Data Engineering '88 Proceedings*, Feb. 1988.

[BL75] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition & Multics Interpretation. Technical Report MTIS AD-A023588, MITRE Corporation, July 1975.

[BN79] J. Beauquier and M. Nivat. Applications of Formal Language Theory to Problems of Security and Synchronization. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 407–454. Academic Press, 1979.

[BN87] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proc. 1989 IEEE Symposium on Research in Security & Privacy*, pages 215–228. IEEE Computer Society Press, April 1987.

[Cat94] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufman, 1994.

[Cha86] L. S. Chalmers. An Analysis of the Differences Between the Computer Security Practices in the Military and Private Sectors. In *Proc. 1986 IEEE Symposium on Research in Security & Privacy*, pages 71–74. IEEE Computer Society Press, April 1986.

[CJ85] J. M. Carroll and H. Jurgensen. Design of a Secure Relational Database. In J. B. Grimson and H. J. Kugler, editors, *The Practical Issues in a Troubled World. Proc. 3rd IFIP Conference on Computer Security*. North Holland. August 1985.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Presss, 1990.

[CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Security Policies. In *Proc. 1987 IEEE Symposium on Research in Security & Privacy*, pages 184–194. IEEE Computer Society Press, April 1987.

[CY92a] F. Cuppens and K. Yazdanian. Logic Hints and Security in Relational Database. In C. E. Landwehr and S. Jajodia, editors, *Database Security V: Status & Prospects*, pages 227–238. North-Holland, 1992.

[CY92b] F. Cuppens and K. Yazdanian. A Natural Decomposition of Multilevel Relations. In *Proc. 1992 IEEE Symposium on Research in Security & Privacy*. IEEE Computer Society Press, May 1992.

[Dat83] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1983.

[Dat86] C. J. Date. *Relational Databases: Selected Writings*. Addison-Wesley, 1986.

[Dat90] C. J. Date. A Contribution to the Study of Database Integrity. In *Relational Database: Writings 1985-1989*, chapter 7, pages 185–215. Addison-Wesley, 1990.

[Day93] U. Dayal. An Activity/Transaction Model for Distributed Multi-Service System. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Proc. Int'l Workshop on Distributed Object Management Systems, University of Alberta, Edmonton, Canada August 1992*, pages 246–251. Morgan Kaufmann, 1993.

[DD77] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Den82] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers & Transactions. In *SIGMOD Proc. 1990 Int'l Conf on Management of Data*, pages 204–214. ACM Press, May 1990.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactions Model for Long-Running Activities. In *Proc. 1991 Int'l Conf on Very Large Data Bases*, pages 113–122, Sept 1991.

[DHP89] K. R. Dittrich, M. Hartig, and H. Pfefferle. Discretionary Access Control in Structurally Object Oriented Database Systems. In C. E. Landwehr, editor, *Database Security II: Status & Prospects*, pages 105–121. North-Holland, 1989.

[Dit90] K. R. Dittrich. Object Oriented Database Management Systems: The Next Miles of the Marathon. *Information Systems*, 15(1):161–167, Mar 1990.

[DM89] J. E. Dobson and J. A. McDermid. Security Models and Enterprise Models. In C. E. Landwehr, editor, *Database Security II: Status & Prospects*, pages 1–39. North-Holland, 1989.

[DS92] D. E. Denning and W. Shockley. Discussion: Pros and Cons of the Various Approaches. In T. F. Lunt, editor, *Research Directions in Database Security*, pages 97–103. Springer-Verlag, 1992.

[ELMB92] A. K. Elmargamid, Y. Leu, J. G. Mullen, and O. Bukhres. Introduction to Advanced Transactions Models. In Ahmed K. Elmargarmid, editor, *Database Transaction Models for Advanced Applications*, pages 33–52. Morgan Kaufmann, 1992.

[Eng87] N. C. K. Eng. A Model for Security in Information Systems. Master's thesis, Department of Computer Science, University of Western Ontario, Canada, 1987.

[FKMT91] E. Fong, W. Kent, K. Moore, and C. Thompson, editors. *X3/SPARC/DBSSG/OODBTG Final Report*. Sept 1991.

[Fol87] S. N. Foley. A Universal Theory of Information Flow. In *Proc. 1987 IEEE Symposium on Research in Security & Privacy*, pages 116–122. IEEE Computer Society Press, April 1987.

[Fol91] S. N. Foley. A Taxonomy for Information Flow Policies. In *Proc. 1991 IEEE Symposium on Research in Security & Privacy*, pages 98–108. IEEE Computer Society Press, April 1991.

[GMP92] J. Glasgow, G. MacEwen, and P. Panangaden. A Logic for Reasoning About Security. *ACM Transactions on Computer Systems*, 10(3):226–264, August 1992.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Hin88] T. F. Hinke. Inference Aggregation Detection in Database Management Systems. In *Proc. 1988 IEEE Symposium on Research in Security & Privacy*. IEEE Computer Society Press, April 1988.

[HKM79] D. K. Hsiao, D. S. Kerr, and S. E. Madnick. *Computer Security*. Academic Press, 1979. ISBN 0-12-357650-4.

[JK90] S. Jajodia and B. Kogan. Integrating an Object-Oriented Data Model with Multilevel Security. In *Proc. 1990 IEEE Symposium on Research in Security & Privacy*, pages 76–85. IEEE Computer Society Press, May 1990.

[JS90] S. Jajodia and R. Sandhu. Polyinstantiation Integrity in Multilevel Relations. In *Proc. 1990 IEEE Symposium on Research in Security & Privacy*. IEEE Computer Society Press, 1990.

[JS91a] S. Jajodia and R. Sandhu. A Novel Decomposition of Multilevel Relations into Single Level Relations. In *Proc. 1991 IEEE Symposium on Research in Security & Privacy*, pages 300–313. IEEE Computer Society Press, 1991.

[JS91b] S. Jajodia and R. Sandhu. Polyinstantiation Integrity in Multilevel Relations Revisited. In S. Jajodia and C. E. Landwehr, editors, *Database Security IV: Status & Prospects*, pages 297–307. North-Holland, 1991.

[Kar88] P. A. Karger. Implementing Commercial Data Integrity with Secure Capabilities. In *Proc. 1988 IEEE Symposium on Research in Security & Privacy*, pages 130–139. IEEE Computer Society Press, April 1988.

[KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In *OOPSLA '86 Proceedings*, pages 406–416, Nov 1986.

[Kim90] Won Kim. Object Oriented Databases: Definitions and Resea..a Directions. *IEEE Trans. on Knowledge and Data Engineering*, 2(3):327-341, Sept 1990.

[KM92] E. V. Krishnamurthy and A. McGuffin. On the Design & Administration of Secure Database Transactions. *ACM SIGSAC Review*, pages 63–70, Spring/Summer 1992.

[KMV+90] S. Krakowiak, M. Meysembourg, H. N. Van, M. Reveill, C. Roisin, and X. R. de Plna. Design and Implementation of an Object Oriented Strongly Typed Language for Distributed Applications. *Journal of Object Oriented Programming*, 3(3):11–22, Sept/Oct 1990.

[LAC+94] M. E. S. Loomis, T. Atwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade. The ODMG Object Data Model. *Journal of Object Oriented Programming*, 7(6):64–69, June 1994.

[Lan81] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3), Sept 1981.

[Lan83] C. E. Landwehr. The Best Available Technologies for Computer Security. *IEEE Computer*, 16(7), July 1983.

[Law93] L. G. Lawrence. The Role of Roles. *Computers & Security*, 12(1):15–21, Feb 1993.

[Lee88] T. M. P. Lee. Using Mandatory Integrity to Enforce "Commercial" Security. In *Proc. 1988 IEEE Symposium on Research in Security & Privacy*, pages 140–146. IEEE Computer Society Press, April 1988.

[LH91] T. F. Lunt and D. Hsieh. Update Semantics for Multilevel Relations. In S. Jajodia and C. E. Landwehr, editors, *Database Security IV: Status & Prospects*, pages 281–296. North-Holland, 1991.

[Lin76] T. A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, Dec. 1976.

[Lin91] T. Y. Lin. Multilevel Databases and Aggregated Algebra. In S. Jajodia and C. E. Landwehr, editors, *Database Security IV: Status & Prospects*, pages 325–350. North-Holland, 1991.

[Lip82] J. S. B Lipner. Non-Discretionary Controls for Commercial Applications. In *Proc. 1982 IEEE Symposium on Research in Security & Privacy*, pages 2–10. IEEE Computer Society Press, April 1982.

[Liu80] L. Liu. On Secure Flow Analysis in Computer Systems. In *Proc. 1980 IEEE Symposium on Research in Security & Privacy*, pages 22–33. IEEE Computer Society Press, April 1980.

[LSS+88] T. F. Lunt, R. R. Schell, W. R. Shockley, M. Heckman, and D. Warren. A Near-Term Design for the Sea-View Multilevel Database Systems. In *Proc. 1988 IEEE Symposium on Research in Security & Privacy*. IEEE Computer Society Press, April 1988.

[Lun89] T. F. Lunt. Aggregation and Inference: Facts and Fallacies. In *Proc. 1989 IEEE Symposium on Research in Security & Privacy*. IEEE Computer Society Press, April 1989.

[Lun92] T. F. Lunt, editor. *Research Directions in Database Security*. Springer-Verlag, 1992. ISBN 0-387-97736-8.

[LVV88] C. Lecluse, P. Velez, and F. Velez. O2 an Object Oriented Data model. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, 1988.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[McH85] J. McHugh. An Information Flow Tool for Gypsy. In *Proc. 1985 IEEE Symposium on Research in Security & Privacy*, pages 46–48. IEEE Computer Society Press, April 1985.

[MHT92] M. Morgenstern, T. Hinke, and B. Thuraisingham. Inference and Aggregation. In T. F. Lunt, editor, *Research Directions in Database Security*, pages 143–159. Springer-Verlag, 1992.

[Mil81] J. K. Millen. Information Flow Analysis of Formal Specifications. In *Proc. 1981 IEEE Symposium on Research in Security & Privacy*, pages 3–8. IEEE Computer Society Press, April 1981.

[Mot89] A. Motro. Integrity = Validity + Completeness. *ACM TODS*, 14(4):480–502, Dec 1989.

[MS90] D. Maier and J. Stein. Development and Implementation of an Object Oriented DBMS. In S. B. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*. Morgan Kaufmann, 1990.

[NO91a] G. Matunda Nyanchama and S. L. Osborn. Mandatory Security in an Object Oriented Database. Technical Report #317, Department of Computer Science, The University of Western Ontario, London Canada, Dec 1991.

[NO91b] G. Matunda Nyanchama and S. L. Osborn. Orthogonal Views in Object Oriented Database Security. Technical Report # 308, Department of Computer Science, The University of Western Ontario, London Canada, March 1991.

[NO93a] M. Nyanchama and S. L. Osborn. Role-Based Security, Object Oriented Databases & Separation of Duty. *ACM SIGMOD RECORD*, 22(4):45–51, Dec 1993.

[NO93b] M. Nyanchama and S. L. Osborn. Role-Based Security: Pros, Cons & Some Research Directions. *ACM SIGSAC Review*, 2(2):11–17, June 1993.

[NO94a] G. Matunda Nyanchama and S. L. Osborn. Database Security Issues in Distributed Object Oriented Databases. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Data Management*, pages 92–97. Morgan Kaufman, 1994. Proc. of Int'l Workshop on Distributed Object Oriented Databases, University of Alberta, Edmonton, Canada August 1992.

[NO94b] M. Nyanchama and S. L. Osborn. Access Rights Administration in Role-Based Security Systems. In J. Biskup, M. Morgenstern, and C. Landwehr, editors, *Database Security VIII: Status & Prospects*, August 1994. To appear.

[NO94c] M. Nyanchama and S. L. Osborn. Information Flow Analysis in Role-Based Security Systems. *"All about nothing", Journal of Computing & Information*, 1(1), May 1994. Special Issue: Proc. of the 6th International Conference on Computing and Information (ICCI), Peterborough, Ontario, Canada.

[NP90] M. J. Nash and K. R. Poland. Some Conundrums Concerning Separation of Duty. In *Proc. 1990 IEEE Symposium on Research in Security & Privacy*, pages 201–207. IEEE Computer Society Press, May 1990.

[NRZ92] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A Cooperative Transaction Model for Design Databases. In Ahmed K. Elmargarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53–85. Morgan Kaufmann, 1992.

[Nya91] Matunda Nyanchama. Object Oriented Database Security. Master's thesis, Department of Computer Science, The University of Western Ontario, London Ontario, N6A 5B7, Canada, August 1991.

[Nya93a] Matunda Nyanchama. Access Rights Administration in Role-Based Protection Systems. In *UWORKS 1993: Graduate Seminar of the Department of Computer Science*, number # 385 Tech Report. Oct 1993.

[Nya93b] Matunda Nyanchama. Commercial Integrity, Roles & Object-Orientation, Oct 1993. Doctoral Dissertation Research Proposal, Department of Computer Science, The University of Western Ontario, London Ontario, N6A 5B7, Canada.

[oC93] Government of Canada. *The Canadian Trusted Computer Product Evaluation Criteria*. Communications Security Establishment, Government of Canada, 1993.

[oD85] Department of Defence. *Department of Defence Trusted Computer System Evaluation Criteria DoD 5200-28-STD*. Department of Defence, Dec 1985. The Orange Book.

[oD87] Department of Defence. *Trusted Network Interpretation, NCSC-TG-005*. US Department of Defence, July 1987. The Red Book.

[Osb89a] S. L. Osborn. Algebraic Query Optimization for an Object Algebra. Tech. Report #251, Department of Computer Science, University of Western Ontario, London Canada, 1989.

[Osb89b] S. L. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. *IEEE Transactions on Knowledge and Data Engineering*, pages 310–317, Sept. 1989.

[PÖS92] R. J. Peters, M. T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query & View Support in Object Database Systems. Technical Report TR92-14, Department of Computer Science, The University of Alberta, Edmonton, Canada, Oct 1992.

[RGN90] T. C. Rakow, J. Gu, and E. J. Neuhold. Serializability in Object-Oriented Database Systems. In *Proc. Sixth Int'l Conf. on Data Eng.*, pages 112–120. IEEE Computer Society Press, Feb. 1990.

[RWBK91] F. Rabitti, D. Woelk, E. Bertino, and W. Kim. A Model of Authorization for Next Generation Databases Systems. *ACM TODS*, 16(1):88–131, March 1991.

[RWK88] F. Rabitti, D. Woelk, and W. Kim. A Model of Authorization for Object Oriented and Semantic Databases. In *Proc. of Int'l Conference on Extending Database Technology*, March 1988.

[San88] R. Sandhu. The NTree: A Two Dimensional Partial Order for Protection Groups. *ACM Trans. on Computer Syst.*, 6(2):197–222, May 1988.

[San89] R. Sandhu. Recognizing Immediacy in an N-Tree Hierarchy and its Applications to Protection Groups. *IEEE Trans. on Software Engineering*, 15(12):1518–1525, Dec 1989.

[San91] R. Sandhu. Separation of Duties in Computerized Information Systems. In S. Jajodia and C. E. Landwehr, editors, *Database Security, IV: Status and Prospects*, pages 179–189. North-Holland, 1991.

[San93] R. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, 26(11):9–19, Nov 1993.

[SD87] R. R. Schell and D. E. Denning. Integrity in Trusted Database Systems. In M. D. Abrams and H. J. Podell, editors, *Tutorial: Computer and Network Security*, pages 202-208. IEEE Computer Society Press, 1987.

[SS77] J. M. Smith and D. C. Smith. Database Abstractions: Aggregation and Generalization. *ACM Trans. on Database Systems*, pages 105–133, June 1977.

[SW92] K. P. Smith and M. S. Winslett. Entity Modeling in MLS. In Li-Yan Yuan, editor, *Proc. of the 18th Int'l Conf. on Very Large Data Bases*, pages 199–210, Aug 1992.

[TDH92] T. C. Ting, S. A. Demurjian, and M. Y. Hu. Requirements Capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model. In C. E. Landwehr and S. Jajodia, editors, *Database Security V: Status & Prospects*, pages 275–296. North-Holland, 1992.

[Tho91] D. J. Thomsen. Role-Based Application Design and Enforcement. In S. Jajodia and C. E. Landwehr, editors, *Database Security, IV: Status and Prospects*, pages 151–168. North-Holland, 1991.

[VB91] V. Varadharan and S. Black. Multilevel Security in a Distributed Object-Oriented System. *Computers & Security*, 10(1):51–68, Feb 1991.

[Weg90] P. Wegner. Concepts and Paradigms of Object Oriented Programming. *OOPS Messenger*, 1:7-87, August 1990.

[Wis90] Simon R. Wiseman. On the Problem of Security in Data Bases. In D. L. Spooner & C. E. Landwehr, editor, *Database Security III: Status & Prospects*, pages 301–310. North-Holland, 1990.

[Wis92] Simon Wiseman. Abstract and Concrete Models for Secure Database Applications. In C. E. Landwehr and S. Jajodia, editors, *Database Security V: Status & Prospects*, pages 239–273. North-Holland, 1992.

[WR92] H. Wacher and A. Reuter. The Contract Model. In Ahmed K. Elmargarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.

[WS92] G. Weikum and Hans-J Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In Ahmed K. Elmargarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[YBK87] W. D. Young, W. E. Boebert, and R. Y. Kain. Proving Computer Systems Secure. In M. D. Abrams and H. J. Podell, editors, *Tutorial: Computer and Network Security*, pages 202-208. IEEE Computer Society Press, 1987.

[ZM90] S. B. Zdonik and D. Maier. Fundamentals of Object Oriented Database Systems. In S. B. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*. Morgan Kaufmann, 1990.